MASTER EN SCIENCES INFORMATIQUES

Méthodes et outils de conception orientés-agent

Trigaux, Jean-Christophe; Vermaut, François

Award date:
2002

FUNDP
Institut d'Informatique
Rue Grandgagnage, 21
B - 5000 NAMUR (Belgique)

Méthodes et outils de conception orientés-agent

Jean-Christophe Trigaux

François Vermaut

Promoteur : Pierre-Yves Schobbens
(FUNDP - Institut d'Informatique, Namur)

Maître de stage : Frank De Boer
(UU - Institute of ICS, Utrecht, The Netherlands)

Mémoire présenté pour l'obtention
du grade de maître en informatique

Septembre 2002

Résumé

Les systèmes multi-agents deviennent de plus en plus populaires, et certains considèrent même qu'un nouveau paradigme est né : le paradigme orienté-agent. Dès lors, il est nécessaire pour ce paradigme de disposer de méthodes et outils afin de simplifier et standardiser le développement de systèmes multi-agents. Ce mémoire a pour but de présenter ces méthodes et outils et de le mettre dans une perspective d'ingénierie logicielle orientée-agent. Du point de vue des méthodes, nous présenterons les méthodologies AAII, Gaia et MAS-CommonKADS, ainsi que les fondements du formalisme BDI pour la spécification de systèmes multi-agents. Du point de vue des outils, nous présenterons les langages de programmation 3APL, AgentSpeak(L) et Congolog. Nous finirons par fournir un exemple concret de développement d'un système multi-agents utilisant certains des outils et méthodes présentés, au travers d'un cas applicatif simple.

Mots-Clés : Systèmes multi-agents (SMA), Ingénierie logicielle orientée-agent, Architecture BDI, AAII, Gaia, MAS-CommonKADS, 3APL, AgentSpeak(L), ConGolog

Abstract

Multiagent systems become more and more popular, and some people even consider that a new paradigm is born : the agent-oriented paradigm. It is thus necessary for this paradigm to have methods and tools in order to simplify the development of multiagent systems. The aim of this thesis is to present these methods and tools and to put these ones from the point of view of the agent-oriented software engineering. Concerning the methods, we shall present the AAII , Gaia and MAS-CommonKADS methodologies and the basis of the BDI formalism for multiagent systems specification. Concerning the tools, we shall present the 3APL, AgentSpeak(L) and Congolog programmation languages. We shall finish with a concrete example of a multiagent system development using some of the methods and tools presented, through a simple applicative case.

Keywords : Multiagent Systems (MAS), Agent-Oriented Software Engineering (AOSE), BDI Architecture, AAII, Gaia, MAS-CommonKADS, 3APL, AgentSpeak(L), ConGolog

Remerciements

Nous aimerions remercier toutes les personnes qui nous aidé durant la rédaction de ce mémoire, et en particulier le Prof. Pierre-Yves Schobbens, notre promoteur.

Nous remercions également le Prof. Frank De Boer de nous avoir si promptement accepté pour notre stage au sein du projet de recherche 3APL à l'université d'Utrecht. Nous remercions aussi MM. Mehdi Dastani et Meindhert Kroese pour leur disponibilité dans le cadre de ce stage, ainsi que Mme Monique Dixon pour nous avoir permis de trouver un logement à Utrecht.

Pour finir, nous aimerions remercier nos parents ainsi que tous nos proches, sans qui l'accomplissement de nos études n'aurait certainement pas été possible.

Table des matières

Table des figures

Introduction

Dans la vie des entreprises, la construction des systèmes informatiques actuels est basée, dans la majorité des cas, sur l'analyse organisationnelle de l'entreprise et sur un cahier des charges établi en accord avec les besoins des utilisateurs à un moment précis. Le système informatique constitue, à lui seul, un risque non négligeable pour ces entreprises en évolution continue, car il peut devenir un frein important à leur propre progression. L'un des problèmes engendrant ce risque est, selon M. Wooldridge [52], l'interaction entre composants, qui constitue la caractéristique principale engendrant la difficulté d'une ingénierie correcte et efficiente du logiciel. Le concept d'agent, par ses mécanismes de coordination et négociation, est donc le plus à même de résoudre ces problèmes. Dès lors, l'ingénierie logicielle basée sur le concept d'agent, nommée "ingénierie logicielle orientée-agent", ou "paradigme orienté-agent", a besoin de méthodes et outils afin de permettre le développement de systèmes informatiques de manière relativement aisée.

L'ingénierie logicielle orientée-agent est composée principalement de deux disciplines. La première est la **conception orientée-agent**, et définit le processus d'élaboration d'un ou plusieurs modèles du système désiré qui puissent être implémentés d'une manière relativement systématique. La conception orientée-agent est traditionnellement divisée en deux classes de méthodes :

- les méthodes informelles qui permettent de guider le développeur à travers les étapes de conception et de maintenance d'un système. Ces méthodes fournissent un support au *processus* de développement de systèmes multi-agent, et permettent ainsi à la conception logicielle orientée-agent de passer du monde de la recherche au monde de l'entreprise. Nous traiterons dans la première partie des deux classes les plus représentatives de ces méthodes : les méthodologies inspirées de la conception orientée-objet (chapitre 2) et les méthodologies inspirées de l'ingénierie des connaissances (chapitre 3) ;

- les méthodes formelles de spécification de systèmes multi-agent, qui seront traitées dans la deuxième partie. Nous y exposerons le formalisme BDI (chapitre 4), qui permet la spécification d'agents en tant que systèmes intentionnels

La deuxième discipline principale de l'ingénierie logicielle orientée-agent est la **programmation orientée-agent**. Cette discipline comprend les travaux de recherches sur des langages de programmation orientée-agent, ainsi que les techniques de programmation orientée-agent. Cette discipline a pour but de permettre l'implémentation directe de spécifications multi-agents. Dans la troisième partie nous présenterons donc trois langages de programmation orientée-agent : 3APL(chapitre 5), AgentSpeak(L) (chapitre 6) et Congolog (chapitre 7).

Dans le cadre de notre stage, nous avons développé la base des croyances 3APL ainsi que de simples outils de communication entre agents 3APL. Vous trouverez la descritpion des principes généraux de cette implémentation dans le chapitre 5 ainsi que le code et la documentation java sur le cd-rom associé à ce mémoire.

Pour finir, nous étudierons un cas concret d'ingénierie logicielle orientée-agent, en fournissant un exemple de développement d'un programme multi-agent à partir de besoins fonctionnels dans la quatrième partie.

Mais avant toute chose, il est important de définir les concepts importants des agents et des systèmes multi-agent, ce que tentera de faire le chapitre 1.

Chapitre 1

Les Systèmes Multi-Agents (SMA)

1.1 Introduction

L'objectif premier des systèmes multi-agents est d'obtenir grâce aux systèmes distribués un comportement cohérent, adaptatif et robuste à partir de l'interaction d'un grand nombre de composants informatiques, éventuellement hétérogènes et non-fiables individuellement [3].

La conception et l'évolution des systèmes distribués sont dues principalement à deux raisons : d'une part, à la plus grande accessibilité des ressources informatiques grâce à l'Internet, et d'autre part, aux réseaux dont l'évolution technologique a amélioré, de manière non négligeable, la communication ainsi que l'échange de données. Désormais, les systèmes distribués doivent faire preuve d'une plus grande flexibilité, c'est à dire être capable d'évoluer dans un environnement changeant et de faciliter l'introduction de nouveaux composants et de nouvelles interconnections.

Toutefois, l'idée première qui consistait à contrôler simplement des modules informatiques à été influencée par la théorie des organisations, car désormais le but est de fournir aux agents les moyens de s'organiser. On va donc parler de systèmes multi-agents dans lesquels les agents devront interagir, coopérer (contrôle), collaborer (allocation de tâches), négocier (résolution de conflits) et se coordonner (synchronisation).

Il est difficile de déterminer le rôle que jouent et que pourront jouer les systèmes multi-agents dans un avenir proche. Dans cette optique, il est nécessaire de définir les différentes notions et frontières de ce domaine pour pouvoir mieux cerner le rôle que l'on essaye de lui faire jouer dans la conception de système informatique, à travers ses méthodologies et ses outils.

Ce premier chapitre a pour objectif de familiariser le lecteur avec les Systèmes Multi-Agents (SMA) et les différentes notions et concepts qui leurs sont liés. La première partie décrira le contexte historique qui a conduit à l'émergence des SMA. Dans un deuxième temps, nous définirons les concepts d'agent et de SMA ainsi que leurs caractéristiques respectives. Sur base de ces notions, nous pourrons identifier les différentes conditions d'applications de cette théorie. Enfin, nous présenterons les domaines d'applications et quelques exemples réels basés sur ces concepts.

1.2 Contexte historique

Au départ, la communauté informatique considérait les systèmes logiciels comme des entités recevant en entrée des données, qu'elles transformaient afin de fournir en sortie des résultats. Cette idée fut à la base du développement des systèmes centralisés et séquentiels.

Mais, dès que la complexité des problèmes à traiter devint plus importante, il est devenu nécessaire de décomposer les systèmes logiciels en plusieurs sous-systèmes moins complexes. Leurs capacités à résoudre des problèmes localisés et à interagir entre eux, permirent de résoudre des problèmes globaux d'une plus grande complexité. De cette approche décentralisée utilisée dans le domaine de l'Intelligence Artificielle (IA) est née l'Intelligence Artificielle Distribuée (IAD). En quelques mots, l'IAD postule que la conception des systèmes intelligents peut se faire grâce à un ensemble d'entités (sous-systèmes) plus primitives qui sont capables d'interagir entre elles.

Les systèmes multi-agents sont donc issus du domaine de l'IAD qui considéra l'interaction des agents comme la clé de voûte des systèmes capables de mener à bien des tâches complexes. L'IAD montra qu'il est possible de générer des solutions à partir de points de vue multiples et parfois contradictoires. Les activités d'un système sont donc considérées comme le fruit des interactions entre divers agents concurrents et autonomes. Ces agents travaillent au sein de communautés selon des mécanismes parfois complexes de coopération, de concurrence et de gestion de conflits qui font émerger des comportements globaux dans le but d'accomplir les tâches du système [13].

L'IAD n'est pourtant pas la seule discipline à la base des SMA. La vie artificielle a contribué au même titre que l'IAD à l'émergence des SMA. Cette combinaison a permis d'apporter une dimension à la fois cognitive et réactive. En effet, l'IAD est attachée au concept d'intelligence en tant que manipulation de symboles, tandis que la vie artificielle attache une importance toute

particulière aux concepts d'autonomie, de comportement et de viabilité.

Dans les faits, on peut trouver l'origine de la technologie des SMA, principalement dans trois domaines.

Tout d'abord, au niveau de *l'intelligence artificielle*, la difficulté qu'il y a à traduire un ensemble d'expertises sous une forme unifiée a amené les chercheurs à développer des systèmes multiexperts, c'est-à-dire des systèmes mettant en jeu plusieurs bases de connaissances plus ou moins coordonnées.

Dans le domaine des *sciences de la vie*, il a été démontré qu'une approche des écosystèmes habités, dans laquelle les individus sont directement représentés sous forme d'entités informatiques, permet d'obtenir une meilleure représentation et un meilleur suivi de leur évolution.

Enfin, au sein de la *robotique*, le développement de la miniaturisation en électronique permet de concevoir des robots qui disposent d'une certaine autonomie quant à la gestion de leur énergie et quant à leur capacité de décision. On a pu alors montrer qu'un ensemble de petits robots ne disposant que de capacités élémentaires de décision et d'intelligence pouvait facilement rivaliser avec les performances d'un seul robot "intelligent", nécessairement plus lourd et plus difficile à gérer.

1.3 Concepts liés aux systèmes multi-agents

1.3.1 Le concept d'agent

Définition d'un Agent

Pour Wooldridge, "un agent est un système informatique capable d'agir de manière autonome et flexible dans un environnement changeant"[52].

Pour Weiss, "un agent est une "entité computationnelle", comme un programme informatique ou un robot, qui perçoit et agit de façon autonome sur son environnement"[49]. Un agent est autonome dans le sens où son comportement est dans une certaine mesure dirigé par sa propre expérience. Cela implique qu'un agent peut percevoir son environnement et agir de manière autonome sur celui-ci.

Un agent est donc :

- une entité autonome pouvant offrir des services ;
- une entité dont le comportement est la conséquence de ses objectifs, de sa perception, de ses représentations, de ses compétences et des communications qu'elle peut avoir avec les autres agents ;

- une entité qui poursuit un objectif individuel, voire une fonction de satisfaction qu'elle cherche à optimiser ;
- une entité qui possède des ressources et des compétences propres ;
- une entité qui est apte à agir sur l'environnement du système auquel il appartient, et/ou à interagir avec les autres agents ;
- une entité qui peut communiquer avec les autres agents ;
- une entité qui est capable de se reproduire.

La structure d'un agent comprend typiquement :

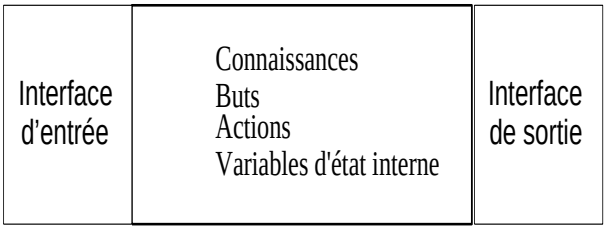


FIG. 1.1 – Structure d'un agent

Une interface d'entrée et une interface de sortie sont nécessaires pour que l'agent puisse communiquer avec son environnement. Un agent est doté de connaissances (connues initialement ou acquises par l'agent), de buts, de variables d'état interne et d'actions (sur l'environnement ou sur lui-même).

Les mécanismes de traitement d'un agent se divisent en :

- des mécanismes de raisonnement qui construisent les différentes actions possibles en fonction des buts et des connaissances de l'agent.
- des mécanismes de décision qui, en fonction des buts et des hypothèses, choisissent la ou les actions les plus appropriées.

Les divers types de raisonnement d'un agent se ramènent à des comportements fondamentaux tels que :

- la prise de décision,
- la révision des croyances,
- la capacité à émettre des hypothèses.

Ce type d'agent est capable d'avoir un comportement à la fois flexible, coopératif et rationnel malgré un environnement changeant et imprévisible.

Caractéristiques d'un Agent

Ce sont ces caractéristiques particulières qui différencient les agents des autres concepts informatiques qui s'en rapprochent, tels que les "objets", les "acteurs", les "modules logiciels" ou bien les "processus parallèles".

Pour Wooldridge, un agent doit être doté de quatre caractéristiques essentielles : l'autonomie, la réactivité, la pro-activité et la capacité sociale [50].

- Autonomie : On entend par autonomie le fait qu'un agent, se trouvant dans un état donné à un instant donné, soit capable de prendre des décisions sans l'intervention de l'homme ou d'autres agents. Cela implique donc qu'il n'est pas dirigé par des commandes introduites par un utilisateur. L'autonomie de l'agent (autonome, semi-autonome ou dépendant) est déterminée par ses capacités de flexibilité et d'adaptation.

- Réactivité : Un agent doit être capable de percevoir son environnement et de maintenir un lien constant avec celui-ci afin de répondre aux changements qui y surviennent.Un agent peut, en conséquence, agir sur son environnement.

- Pro-activité : Un agent ne doit pas seulement être dirigé par les événements générés par son environnement, il doit pouvoir prendre des initiatives, il doit faire preuve d'un comportement établi suivant ses propres objectifs.

- Capacité sociale : L'agent doit être vu comme un être social. L'agent, en tant qu'entité individuelle, est intégré à un collectif, à un système dans lequel la distribution des compétences, des tâches et des rôles nécessitent des processus de coordination, de communication, d'organisation et de coopération. Les interactions et les échanges de messages ou de ressources entre les agents sont alors impératifs. Pour un agent, interagir avec un autre constitue à la fois la source de sa puissance et l'origine de ses problèmes. C'est en effet parce qu'ils coopèrent que des agents peuvent accomplir plus que la somme de leurs actions individuelles, mais c'est aussi à cause de leur multitude qu'ils doivent coordonner leurs actions et résoudre des conflits.

En outre, on attribue parfois aux agents des caractéristiques particulières, notamment des caractéristiques semblables à celles des humains, afin de modéliser au mieux le comportement des agents. Des caractéristiques comme la connaissance, les croyances, l'intention et l'obligation sont des exemples de

notions mentales normalement constatées chez l'homme qui sont éventuellement utilisées dans la conception d'agents [45].

Typologie des agents

Il y a deux grandes écoles de pensée dans la communauté multi-agents, chacune concevant les agents de manières différentes. La première, l'école cognitive, conçoit les agents comme des entités déjà "intelligentes", c'est-à-dire capables de résoudre certains problèmes par eux-mêmes. La deuxième école, la réactive, conçoit les agents comme des entités très simples réagissant directement aux modifications de l'environnement [50].

Les agents cognitifs ou agents intelligents, en raison de leur sophistication et de leur capacité de raisonnement, peuvent travailler relativement indépendamment sur certaines tâches. Pour arriver à leur objectif, ils doivent à la fois coordonner leurs activités et gérer certaines négociations afin de pouvoir satisfaire au mieux leur(s) objectif(s). Cette capacité offre une grande souplesse dans l'expression du comportement des agents, par contre ses limites sont mises en évidence par la difficulté de résoudre les conflits liés aux incompatibilités des buts des agents du système. Cette vision des agents correspond évidement aux concepts introduits par l'IAD au niveau des SMA. Les agents cognitifs sont caractérisés par le fait qu'ils ont une mémoire du passé, qu'ils sont capables de planification, d'engagement et qu'ils ont un mode d'organisation sociale.

Les agents réactifs, contrairement aux cognitifs, n'ont pas une intelligence individuelle ; ils sont calqués sur les modèles de la vie artificielle. Ils n'ont pas de mémoire du passé et suivent un mode d'organisation biologique. C'est pourquoi ils possèdent de simples mécanismes de réaction aux événements et n'agissent qu'à travers les stimuli fournis par l'environnement ou créés à partir d'autres agents. Ce n'est pas l'agent en lui-même qui est considéré comme intelligent, mais c'est le système qui est capable, globalement, d'avoir un comportement intelligent.

Pour illustrer ce concept, on peut comparer ce type d'organisation à celui d'une colonie de fourmis. En effet, toutes les actions des membres d'une colonie sont régies par un objectif commun qui est la survie de la colonie. L'ensemble des fourmis est capable de réagir de manière efficace par rapport à des problèmes conséquents comme la recherche de nourriture, la gestion des larves et des oeufs, la constructions des nids, ... La somme des membres de la colonie est capable d'actions évoluées, mais chaque individu pris séparément possède une représentation faible de l'environnement et n'a pas

de buts globaux. Les activités complexes déterminées par un but sont, dans cette architecture d'agents, une propriété émergente des interactions sociales entre les agents constituant le groupe.

La distinction entre cognitif et réactif n'est pourtant pas toujours pertinente et/ou évidente. En effet, il existe d'autres types d'agents tel que les agents intentionnels qui sont doués d'une intelligence individuelle et qui possèdent explicitement des buts motivant leurs actions. Ils sont alors capables de concevoir des plans et de prévoir des réactions possibles à leurs actions en vue d'accomplir ces buts. Cette capacité d'anticipation et de planification permet à ce type d'agent d'optimiser son comportement et de n'effectuer ainsi que les actions véritablement nécessaires.

Ferber [13] distingue principalement quatre types d'agents suivant le type de relations (cognitif ou réactif) qu'ils ont avec leur environnement et leur comportement individuel :

1. Intentional Agents :
   Un agent intentionnel est un agent de type cognitif dont le comportement est régi par des objectifs explicites. Un agent intentionnel peut tout à fait réagir à des stimuli de son environnement tout en conservant une haute représentation de celui-ci et en agissant en fonction de ses intentions. Au sein de l'architecture des agents intentionnels, on sépare les agents dont l'intentionnalité est immédiate de ceux dont l'intentionnalité est dirigée vers le futur. Dans le premier cas, l'intentionnalité représente un mécanisme d'exécution des actes de l'agent, alors que, dans le deuxième cas, elle fait intervenir une planification des actes de l'agent.

2. Module-based Agents :
   Un agent "Module-based" est un agent de type cognitif dont le comportement est régi par des stimuli, des réflexes.

3. Drive-based Agents :
   Un agent "Drive-based" est un agent de type réactif dont le comportement est régi par des objectifs explicites.

4. Tropistic Agents :
   Un agent "Tropistic" est un agent de type réactif dont le comportement est régi par des stimuli, des réflexes.

Distinction entre les notions d'agent et d'objet

Les paradigmes de programmation orientés agents et objets peuvent parfois paraître fort proches. Souvent les notions d'agent et d'objet portent à confusion. Afin de permettre une meilleure compréhension, nous pouvons les distinguer en identifiant quelques une de leurs similarités et différences majeures.

Tout comme les agents, les objets sont caractérisés par leur comportement, l'état dans lequel ils se trouvent et le fait qu'ils communiquent entre eux par simple échange de messages.

La première différence se situe au niveau du contrôle de leur comportement. En effet, les objets n'ont aucun contrôle sur leur comportement, tandis que les agents le contrôlent, soit de manière réactive, soit pour être en adéquation avec leurs buts.

La seconde différence concerne la notion du comportement autonome et flexible. L'utilisation des méthodes publiques et privées sur les objets implique un manque d'autonomie de ceux-ci puisqu'ils n'ont aucun contrôle sur l'application de ces méthodes. Au contraire les agents sont capables de décider si ils appliquent, ou non, les requêtes qui leur sont demandées, ils doivent raisonner pour atteindre des buts déterminés. Les agents n'invoquent pas de méthodes entre eux mais s'envoient plutôt des requêtes d'exécutions. Un objet est défini par un certain nombre de services (ses méthodes) qu'il ne peut refuser d'exécuter si un autre objet le lui demande et les messages sont donc nécessairement des invocations de méthodes. Le développeur d'un logiciel "objet" doit donc vérifier que tous les objets recevront bien des ordres sensés, qu'ils seront effectivement capables d'exécuter. Par rapport aux objets, les agents peuvent recevoir des messages qui ne sont pas uniquement des demandes d'exécution, mais aussi des informations ou des demandes d'informations sur leurs capacités, etc.

La notion de flexibilité est étroitement liée aux caractéristiques de réactivité, de pro-activité et de capacité sociale définies plus haut. Le modèle standard de programmation orientée-objet ne gère absolument pas ce type de comportement, un objet ne possède pas à la base de telles caractéristiques, même si il est possible de les simuler. Les agents sont programmés suivant des règles de comportement qui lui permettent d'établir et de choisir quelle réaction est la plus adéquate pour une situation donnée. L'élément le plus intéressant est qu'il peut lui-même modifier ses propres règles de comportement.

La troisième différence concerne l'activité des agents et des objets. Les agents, à l'intérieur d'une boucle infinie, observent leur environnement, mettent

à jour leur état, sélectionnent et exécutent des actions de manière continue. Tandis que les objets ne deviennent actifs que lorsqu'un autre objet invoque une de leurs méthodes.

La dernière différence montre que les agents sont capables de se tromper : en effet si les croyances d'un agent sont erronées, il peut faire un mauvais choix. Cette capacité de faire des erreurs est un point très positif, car les agents peuvent désormais apprendre par eux-mêmes en modifiant leurs croyances et leurs règles de comportement. Un objet n'est pas capable d'avoir ce type de réaction : les erreurs commises sont des erreurs de programmation ou de conception dont il ne peut tirer aucune conclusion.

1.3.2 Le concept de SMA

Définition

Un SMA est un ensemble d'agents "intelligents" qui interagissent dans un environnement commun, afin soit de poursuivre un ensemble de buts, soit de réaliser un ensemble d'actions [50].

Les SMA se sont principalement basés sur l'idée qu'il est possible de représenter sous forme d'algorithmes informatiques le comportement des individus d'un système. Chaque individu est conçu comme un composant logiciel disposant de sa propre autonomie.

Ferber [13] définit de manière plus formelle un SMA comme un système comprenant un certain nombre d'éléments :

- Un Environnement E.
- Un ensemble d'Objets O, ces objets étant situés dans E.
- Un objet est donc associé à une position dans E et peut être modifié, créé, détruit, perçu par les agents du système.
- Un ensemble d'Agents A qui sont des objets particuliers représentant des entités actives du système.($A \subseteq O$)
- Un ensemble de relations R qui lient les objets (et donc les agents) entre eux.
- Un ensemble d'Opérations Op qui permettent aux agents de créer, de percevoir, d'utiliser les objets.
- Des opérateurs qui représentent les applications de ces opérations et leurs impacts sur l'environnement.

Dans la figure 1.2, un exemple général d'un SMA est présenté ; l'accent est essentiellement mis sur le fait que chaque agent a une représentation propre

d'un sous-ensemble de l'environnement et qu'il possède la capacité d'interagir, de communiquer avec celui-ci, dans le but de satisfaire son objectif.

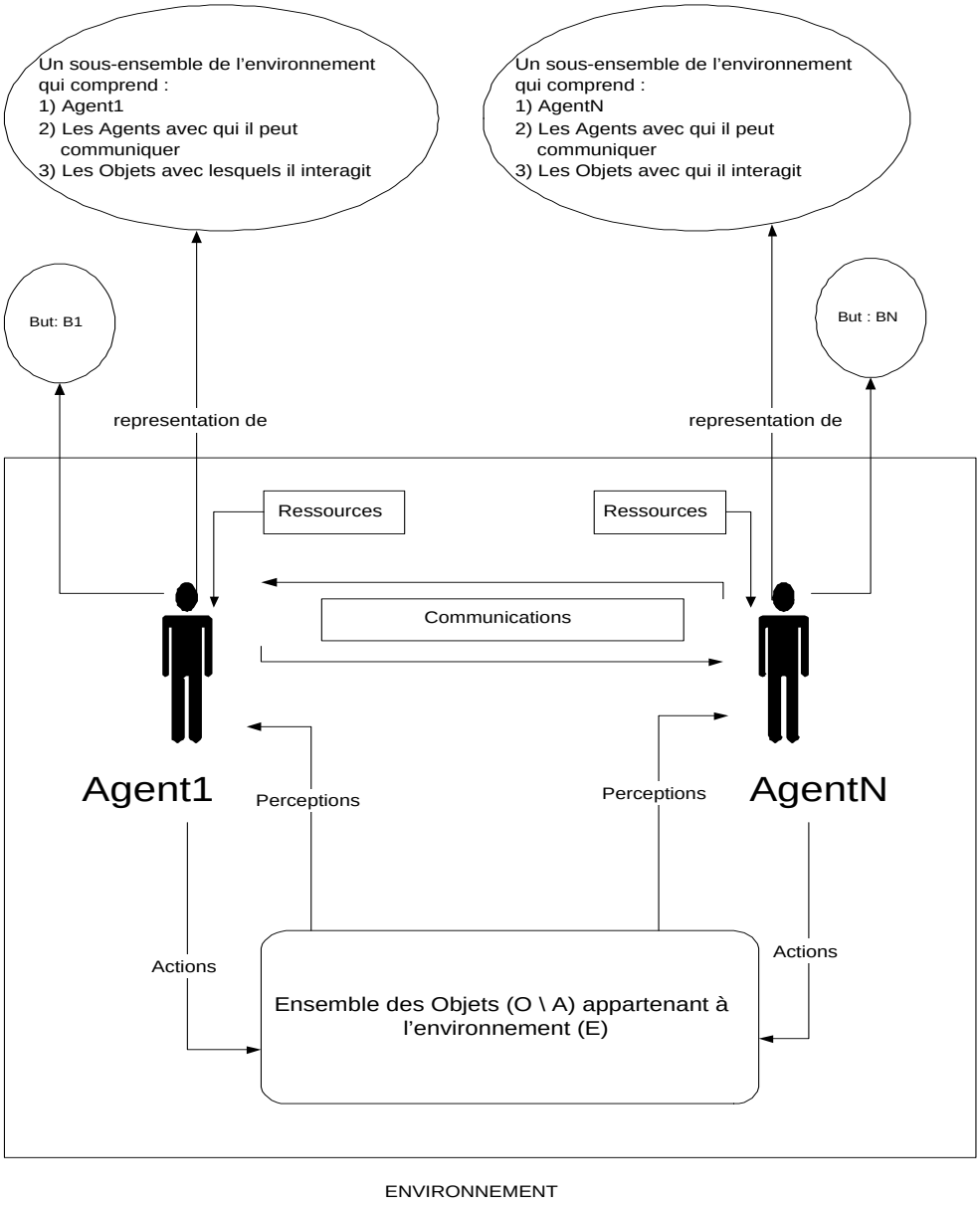


FIG. 1.2 – Schéma d'un SMA

Caractéristiques

Premièrement, un SMA n'est pas dédié à la résolution d'un problème particulier. Un ensemble d'agents autonomes partagent des connaissances et/ou des techniques mais n'ont pas d'organisation préalable. Les agents raisonnent sur leur organisation en fonction du ou des problèmes à traiter.

Deuxièmement, les agents qui font partie d'un SMA ne doivent pas nécessairement connaître le but global pour agir, ils sont autonomes et peuvent réagir seuls par rapport au contexte.

Troisièmement, les données au sein d'un SMA sont décentralisées c'est-à-dire réparties entre les différents agents.

Quatrièmement, un SMA est caractérisé par la coordination, la communication et les relations qui existent entre ses différents agents. La coordination peut s'opérer soit comme une coopération pour aboutir à un but commun, soit comme une négociation pour satisfaire de manière optimale les intérêts de chaque agent. Les conflits d'intérêts peuvent mener à des deadlocks qu'il faut pouvoir gérer via des mécanismes appropriés. La communication s'effectue au moyen de différents protocoles (KQML, FIPA, ...) définissant différents types de messages et de syntaxes.

Selon Ferber : " Les systèmes multi-agents apportent bien plus que des découpages de connaissances : ils renouvellent notre façon d'aborder un problème et de concevoir ce qu'est le raisonnement et l'intelligence. Le problème ne se résout plus en partant d'un état initial pour arriver à un état final, mais en construisant au fur et à mesure des solutions partielles qui se trouvent sur le chemin, chaque agent cherchant à apporter des éléments de solution au fur et à mesure de l'exploration " [12].

En outre, de par leur nature répartie les SMA offrent des propriétés intéressantes liées à leur décentralisation, comme la flexibilité, la robustesse, la sûreté, la réutilisabilité, l'adaptabilité et l'efficacité.

1.4 Conditions d'applications des théories SMA

L'utilisation des SMA n'est ni la solution miracle, ni la solution adéquate à toutes les situations. Cette méthode de développement est pertinente, dans la majorité des cas, lorsque :

- on rencontre des problèmes de nature complexe qui doivent être décomposés ;
- on a besoin de résoudre le problème, et donc les sous-problèmes en découlant, de manière concurrente et/ou simultanée ;
- on a besoin de pouvoir s'adapter à un *environnement changeant* ;
- on a besoin de pouvoir gérer facilement l'ajout ou la modification de composants ;
- différents domaines de connaissances existent ;
- on modélise des systèmes sociaux ou naturels ;
- on distribue des données, du contrôle, des ressources.

On peut souligner également que l'utilisation des SMA entraîne, d'une part, l'absence d'une vision générale et, d'autre part, la difficulté de contrôler de manière globale le système.

1.5 Exemples d'applications utilisant la technologie des SMA

Les théories liées aux systèmes multi-agents s'appliquent principalement aux télécommunications, à l'Internet, au commerce électronique, à des agents physiques tels que les robots, à l'optimisation des systèmes de transports et à la gestion de filières.

La simulation de société ou d'organisation est une des applications les plus pertinentes. En effet, on peut simuler la création et le fonctionnement des structures collectives à partir d'agents qui échangent des informations, des services ou des biens et qui élaborent des contrats. On peut donc simuler de manière beaucoup plus adéquate la société évoluant dans un environnement dynamique influencé par les actions et les interactions entre différents agents.

Pour en avoir un meilleur aperçu et montrer la diversité des domaines d'applications, voici quelques exemples de systèmes basés sur les SMA.

1. Système SIMPOP (Bura) :
   Le modèle développé dans le cadre de ce projet, SIMPOP, est le fruit d'une collaboration étroite entre spécialistes des systèmes urbains et spécialistes de l'intelligence artificielle distribuée et des systèmes multi-agents. L'objectif du modèle est de simuler le passage d'un peuplement en villages à un système de villes hiérarchisé c'est-à-dire de reproduire l'émergence et l'évolution d'un système de villes sur une longue durée (2000 ans). Les systèmes multi-agents offrent des possibilités de conception intéressantes pour la modélisation dynamique des systèmes de villes, permettant de simuler des changements qualitatifs de structure ainsi que de prendre en compte l'espace de façon explicite. Chaque entité spatiale est représentée par un agent informatique qui gère un ensemble de règles de croissance intégrant tout à la fois des dimensions hiérarchique et spatiale.

   http://www.cerma.archi.fr/inventur/inventur064.html

2. Robocup Rescue :
   Dans le domaine de la simulation multi-agents, la Robocup Rescue tire son origine de l'inefficacité des secours actuels en cas de désastre naturel important, comme cela a été illustré lors de la catastrophe de Kobé au Japon (où un tremblement de terre à causé la mort de plus de 6000 personnes). Le but de ce concours est à terme de pouvoir améliorer l'efficacité des secours lors d'une catastrophe civile majeure. Les systèmes multi-agents semblent être la meilleure des solutions pour pouvoir simuler la gestion d'un grand nombre d'agents hétérogènes dans un environnement hostile. Cette simulation permet, entre autre, de créer des outils d'aide aux décisions d'urgence, de prédiction et de planification. Cette simulation, basée sur des photos aériennes, consiste à gérer un ensemble hétérogène d'agents intelligents comme le commandement, les pompiers, les victimes, les policiers, etc. Chacun de ces agents ayant des capacités propres comme la recherche de victimes, le sauvetage de victimes, le déblaiement des voies d'accès et des buildings, l'extinction des incendies, etc.

   Au sein des Facultés Universitaires Notre Dame de la Paix de Namur, ce projet a été étudié de manière beaucoup plus approfondie par Emmeline Leconte, Hugues Van Peteghem et Emmanuel Koch dans le cadre de leur mémoire.

   http://robomec.cs.kobe-u.ac.jp/robocup-rescue/

3. Projets du laboratoire de recherche DAMAS, Dep. d'Informatique, Université LAVAL
   - NetSA (Marc Côté, Nader Troudi et Brahim Chaib-draa)
     "L'évolution de l'Internet et l'apparition des entrepôts de données couplées à la nature dynamique et hétérogène de l'information font en sorte qu'il est de plus en plus difficile de trouver l'information récente recherchée malgré son abondance. Une approche prometteuse à la résolution de ce problème consiste à utiliser des agents logiciels qui coopèrent pour trouver la réponse à une requête d'information. Le projet NetSA (Networked Software Agents) est une architecture de systèmes multi-agents pour la recherche d'informations dans des sources hétérogènes et distribuées."

     http://damas.ift.ulaval.ca/projets/netsa/index.html

   - Auction (Houssein Ben-Ameur et de Brahim Chaib-draa)
     Le commerce électronique devient de plus en plus important car il permet l'automatisation des transactions ce qui diminue considérablement les coûts. Le projet Auction (Enchères multi-objets pour la négociation automatique et le commerce électronique) poursuit cette tendance en testant et validant des protocoles et autres stratégies d'enchères définies dans le contexte des systèmes multi-agents.

     http://damas.ift.ulaval.ca/~benameur/auction/index.html

   - Frégate Canadienne (Brahim Chaib-draa et Sébastien Paquet)
     "Le but de ce projet vise à étudier les systèmes multi-agents pour l'implémentation d'un système d'aide à la décision affecté à la gestion des ressources à bord d'une frégate canadienne de classe HALIFAX. L'étude de ces frégates a été initiée au travers d'un projet fait en collaboration avec la compagnie Lockheed Martin Canada (LMC) et le Centre de Recherches pour la Défense de Valcartier (CRDV). En d'autres mots, le but du système est principalement de gérer toutes les ressources (armes, radars, systèmes électroniques, etc.) disponibles sur un navire de guerre, de type frégate, de manière à augmenter ses chances de survie lors d'attaques par des missiles aériens."

     http://damas.ift.ulaval.ca/projets/TeamWork/

4. Système Guardian (HAYES-ROTH)
   Le système GUARDIAN a pour but de gérer les soins post-opératoires des patients d'une unité chirurgicale de soins intensifs. En effet, le système de fonctionnement d'une unité chirurgicale tourne autour de la bonne coopération entre différents médecins spécialistes et infirmières travaillant dans des domaines distincts. Le partage de l'information entre ces agents est donc capital pour assurer un traitement efficace au patient. Chaque rôle important ou département est donc représenté par un agent à part entière qui doit faire preuve de coopération lorsqu'il veut utiliser les services d'autres agents au sein de l'entreprise.

   http://smi-web.stanford.edu/projects/guardian.html

5. Système ADEPT (O'BRIEN et Jennings)
   ADEPT ou Advanced Decision Environment for Process Tasks est un projet dont l'objectif est de développer un système informatique afin d'assister les entreprises dans leur processus d'affaires. Ce système prend comme principe que le processus d'affaires est un ensemble d'agents qui négocient et qui offrent des services. Au final, les gestionnaires de l'entreprise devraient avoir une meilleure connaissance des informations provenant de leurs départements et, donc, mieux adapter leurs prises de décision en fonction de leurs stratégies.

   http://www.elec.qmul.ac.uk/dai/projects/adept/

   http://citeseer.nj.nec.com/jennings96adept.html

6. Système ARCHON :
   ARCHON (Architecture for Cooperative Heterogeneous ON-line systems) est, au niveau européen l'un des plus grands projets dans le domaine des SMA. Il combine à la fois une architecture générale, software framework et une méthodologie utilisée comme support au développement de SMA dans certains domaines industriels réels. Deux applications de ce projet ont été entreprises, au niveau de la gestion de la distribution d'électricité et du contrôle d'accélérateur de particules. Ces deux tentatives se sont parfaitement déroulées au sein des organisations (respectivement, Iberdrola un distributeur d'électricité dans le nord de l'Espagne et CERN le centre européen de recherche physique

près de Genève).

http://www.ecs.soton.ac.uk/~nrj/archon/test_1.html

7. Système KASBAH (CHAVEZ) :
   Dés 1996, dans le cadre de l'e-commerce est apparu le système KASBAH de CHAVEZ. Kasbah est un marché virtuel sur le Web où les utilisateurs créent des agents autonomes pour acheter et vendre des marchandises en leur nom. L'utilisateur peut paramètrer son agent et définir son comportement afin de le guider dans ses actions.

   http://citeseer.nj.nec.com/201210.html

Première partie

Méthodes informelles de conception orientée-agent

Chapitre 2

Méthodologies inspirées de l'orienté-objet

La plupart des méthodes informelles de conception orientée-agent se basent sur des méthodologies développées par le passé et qui ont déjà fait leurs preuves. Tel est le cas des méthodes présentées dans ce chapitre, qui s'inspirent des méthodologies orientées-objet.

Les méthodologies OO présentent l'avantage de constituer un paradigme qui est arrivé à un haut degré de maturité [32], et dont l'efficacité est universellement reconnue. Une large communauté de développeurs est maintenant familiarisée avec ces méthodologies, c'est pourquoi plusieurs méthodologies orientées-agent se basent sur ces méthodologies orientées-objet.

Toutefois, les méthodologies OO ne sont pas directement applicables aux systèmes multi-agent, étant donné que les agents sont significativement plus complexes que les objets. Il est donc important d'adapter et d'étendre ces méthodologies OO, afin de les rendre applicables aux systèmes multi-agent. La suite de cette section présente les principaux travaux faits en ce sens.

2.1 Méthodologie AAII

Traditionnellement, les méthodologies OO consistent à identifier les différentes classes du domaine d'application, le comportement des objets appartenant à ces classes ainsi que leurs interrelations [32]. Ainsi, les différents détails du système peuvent être capturés dans trois types de modèles, les deux derniers servant à affiner le premier [2, 43] :

- un modèle objet, définissant les informations sur les objets du système, leur structure de données, leurs relations, et leurs opérations

- un modèle dynamique, qui décrit les états, les transitions, les événements, les actions, les activités et les interactions qui caractérisent le comportement du système
- un modèle fonctionnel, qui décrit les flux de données qui ont lieu durant l'activité du système, aussi bien à l'intérieur des composants qu'entre ceux-ci

La méthodologie proposée par Kinny, Georgeff et Rao de l'Australian Artificial Intelligence Institute (AAII) [32], considère que pour spécifier un système multi-agent, il est préférable d'adopter un ensemble de modèles plus spécialisés, qui opèrent à deux niveaux d'abstraction :

- du point de vue *externe*, le système est décomposé en agents, caractérisés par leur finalité, leurs responsabilités, les services qu'ils rendent, les informations dont ils disposent et leurs interactions externes
- du point de vue *interne*, chaque agent possède une architecture propre, constituée dans ce cas-ci de croyances, de buts et de plans

2.1.1 Point de vue externe

Le critère principal de décomposition de cette méthodologie est le critère de *rôle*. L'identification des différents rôles d'un système ainsi que leurs relations permet d'obtenir une hiérarchie de *classes d'agent*, les agents étant des instances de ces classes. Ensuite, à partir de ces rôles, on peut définir les *responsabilités* de chaque classe d'agent, les *services* qui permettent de mettre en oeuvre ces responsabilités et par là même, ses interactions externes. Tout ceci est défini grâce à deux modèles :

- le modèle d'agents : ce modèle décrit les relations hiérarchiques entre classes d'agents (abstraites ou concrètes), ainsi que les différentes instances d'agents qui peuvent exister dans le système
- le modèle d'interaction : ce modèle décrit les responsabilités d'une classe d'agent, les services qu'elle procure et les interactions associées. Ce modèle inclut également la syntaxe et la sémantique des messages échangés entre agents, mais toutefois, la méthodologie AAII ne fournit pas de procédure pour définir ces actes de discours, étant donné le foisonnement des recherches dans ce domaine

Ces deux modèles externes sont largement indépendants de l'architecture BDI, ce qui n'est pas le cas des modèles internes, qui s'en inspirent directement.

2.1.2 Point de vue interne

Dans le paradigme BDI, les agents sont considérés comme ayant certaines attitudes mentales qui sont les Croyances, les Désirs et les Intentions (Beliefs, Desires, Intentions). Comme la méthodologie AAII se base sur ce paradigme BDI, il y a trois modèles internes à un agent (ou plus précisément à une classe d'agents) qui sont les suivants :

- *le modèle de croyances*, qui décrit les informations dont disposent les agents d'une classe sur l'environnement et sur leur état interne. Ce modèle est composé de premièrement, l'*ensemble de croyances* (**belief-set**) qui définit l'ensemble des croyances possibles pour un agent ; deuxièmement, un ensemble d'*états de croyances* (**belief-state**), un *état de croyance* étant une instance particulière de l'*ensemble de croyances*. L'ensemble d'états de croyances définit par exemple les états initiaux possibles
- *le modèle de buts* qui décrit l'ensemble des buts que peut adopter un agent d'une classe. Comme pour le modèle de croyances, ce modèle est composé de deux parties : l'*ensemble de buts* (**goal-set**), et d'un ensemble d'*états de buts* (**goal-state**) spécifiant les états mentaux possibles de l'agent. Le concept de but est l'équivalent dans ce cas-ci du concept de désir
- *le modèle de plans* qui définit la manière avec laquelle on peut résoudre les buts.

Il paraît évident que ces différents modèles interagissent. Les buts sont en effet mis en oeuvre grâce à des plans, et ceux-ci peuvent être déclenchés à partir de certaines croyances dont dispose l'agent à un moment donné. De plus, ces plans modifient généralement les croyances de l'agent, et lui font adopter de nouveaux buts.

2.1.3 Méthode d'élaboration générale de ces différents modèles

1. identifier les différents *rôles* du domaine d'application. Cette identification débouche sur l'élaboration d'une hiérarchie de classes d'agent, et constitue donc le *modèle d'agent* initial ;

2. analyser les différentes *responsabilités* associées à ces *rôles*, et les *services* utilisés pour remplir ces responsabilités. La mise en valeur de ces services permet de dégager les interactions entre différents agents ainsi que les interactions des agents avec l'environnement, ce qui débouche sur la réalisation du *modèle d'interaction*. La définition des services permet également de définir les *buts*, qui seront la base de l'élaboration des modèles internes. Généralement, un ou plusieurs buts seront associés à un service ;
3. l'ensemble des buts dégagés au point précédent constituent le *modèle de buts* ;
4. décomposer chacun des buts en un ensemble de sous-buts et d'actions, en analysant leur ordre, ainsi que les conditions de leur exécution, ce qui débouche sur le *modèle de plans* ;
5. définir la structure de croyances du système, c'est-à-dire les informations nécessaires pour chaque but, sous-but et activité d'un plan. Cette définition débouche sur le *modèle de croyances*. Les prédicats du modèle de croyances permettent désormais de définir de manière formelle les modèles de buts et de plans ;
6. affiner la hiérarchie de classe d'agent à la lumière des nouveaux modèles dont on dispose, en regroupant par exemple les informations et les services communs en de nouvelles classes d'agents

Nous allons maintenant examiner les caractéristiques des différents modèles élaborés, en l'occurrence les modèles d'agents, de croyances, de buts et de plans. Comme nous l'avons précisé précédemment, nous n'allons pas décrire le modèle d'interactions, car la technique utilisée pour ce dernier est laissée au choix du concepteur.

2.1.4 Le modèle d'agents

Le modèle d'agents a deux composants : le *modèle de classes d'agents* et le *modèle d'instances d'agents*, représentés respectivement par un diagramme de classes et un diagramme d'instances. Notons que dans le cas où le nombre de classes d'un système est faible, ces deux diagrammes peuvent être regroupés.

Modèle de classes d'agents

Un modèle de classes d'agents est un graphe dirigé, acyclique, dont les noeuds représentent des classes d'agents abstraites ou concrètes. La relation d'héritage est représentée par un triangle pointant vers la superclasse, la

relation d'agrégation est représentée par un diamant adjacent à la classe agrégée. Les classes d'agent abstraites sont représentées avec un A dans leur coin supérieur droit.

La figure 2.1 représente un modèle de classes et d'instances d'agents pour un système de gestion du trafic aérien [32].

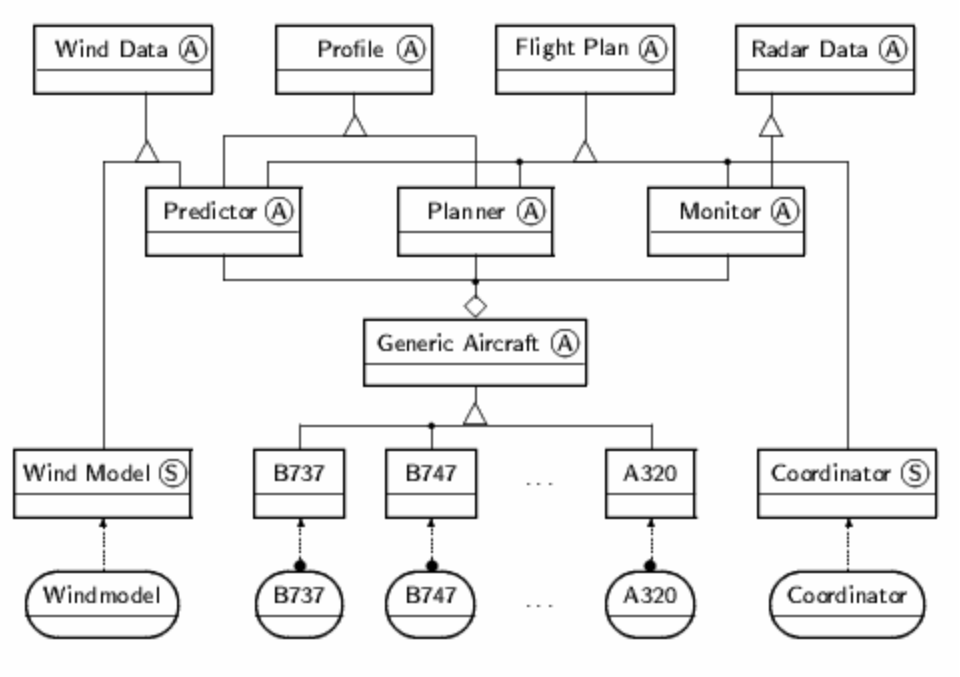


FIG. 2.1 – Modèle de classes et d'instances d'agents.

Chaque classe d'agent peut avoir des attributs, qui sont les modèles de croyances, de plans et de buts.

L'héritage multiple est permis. Les agents héritent des modèles de croyances, de plans et de buts de leurs superclasses, et peuvent affiner ceux-ci. L'agrégation consiste à regrouper en un agent un ensemble de sous-agents, ces sous-agents n'ayant pas accès à leurs croyances, plans et buts respectifs.

Les classes d'agents peuvent prendre quatre autres attributs :

- l'ensemble des *états de croyance* (**belief-state-set**) et l'ensemble des *états de but* (**goal-state-set**) qui définissent l'ensemble des états mentaux initiaux possibles. Ainsi, une instance d'une classe d'agents devra "choisir" son état initial dans ces ensembles
- l'état de croyance *initial* (**initial-belief-state**) et l'état de but *initial* (**initial-goal-state**), c'est-à-dire l'état mental initial par défaut, dans le cas où l'instance de la classe ne définit pas celui-ci

Modèle d'instances d'agents

Le modèle d'instances d'agents est un diagramme d'instances qui définit les agents statiques (créés au moment de la compilation) et les agents dynamiques. Les premiers sont représentés par un S en haut à droite de l'entité. La relation d'instantiation est représentée par une flèche dont l'origine est l'instance de la classe.

Une instance d'agent possède deux attributs : l'état de croyance initial (**initial-belief-state**) et l'état de but initial (**initial-goal-state**), qui sont des valeurs particulières des ensembles d'états de croyance et de but de la classe. Si ces deux attributs ne sont pas définis, ils sont automatiquement hérités par la classe d'agent dont l'instance fait partie.

2.1.5 Le modèle de croyances

Comme nous l'avons dit précédemment, le modèle de croyance d'une classe d'agent est composé d'un *ensemble de croyances* et d'un ou plusieurs *états de croyance*. L'*ensemble de croyances* est spécifié par un ensemble de diagrammes de classes qui définissent le domaine des croyances d'une classe d'agents. Un *état de croyance* est spécifié par un ensemble de diagrammes d'instances qui définissent une instance particulière de l'*ensemble de croyance*. Intuitivement, l'ensemble de croyances est à l'état de croyances, ce que la variable est à la valeur. Ainsi, à un seul ensemble de croyances peut correspondre plusieurs états de croyances, représentants les "valeurs" possibles de l'ensemble de croyances. Toutefois, lorsque l'on définit le modèle de croyances d'une *instance* d'agent, on ne définit bien sûr qu'un seul état de croyance.

Un *ensemble de croyances* est un ensemble de prédicats dont les arguments sont des termes appartenant à un univers de symboles prédéfinis ou définis par l'utilisateur. Ces prédicats sont directement dérivés des définitions de classes de croyances dans le diagramme de l'ensemble des croyances. La figure 2.2 montre un exemple de classes de croyances, avec les prédicats et fonctions dérivés de ces classes. Il est important de préciser que ces classes de croyances n'ont pas la même sémantique que les classes définies dans le modèle d'agents (ou dans un modèle OO par exemple). En effet, une classe de croyances ne constitue pas une entité opérationnelle ayant un certain comportement, elle consiste plutôt en une classe de "concepts" sur lesquels un agent peut avoir certaines croyances.

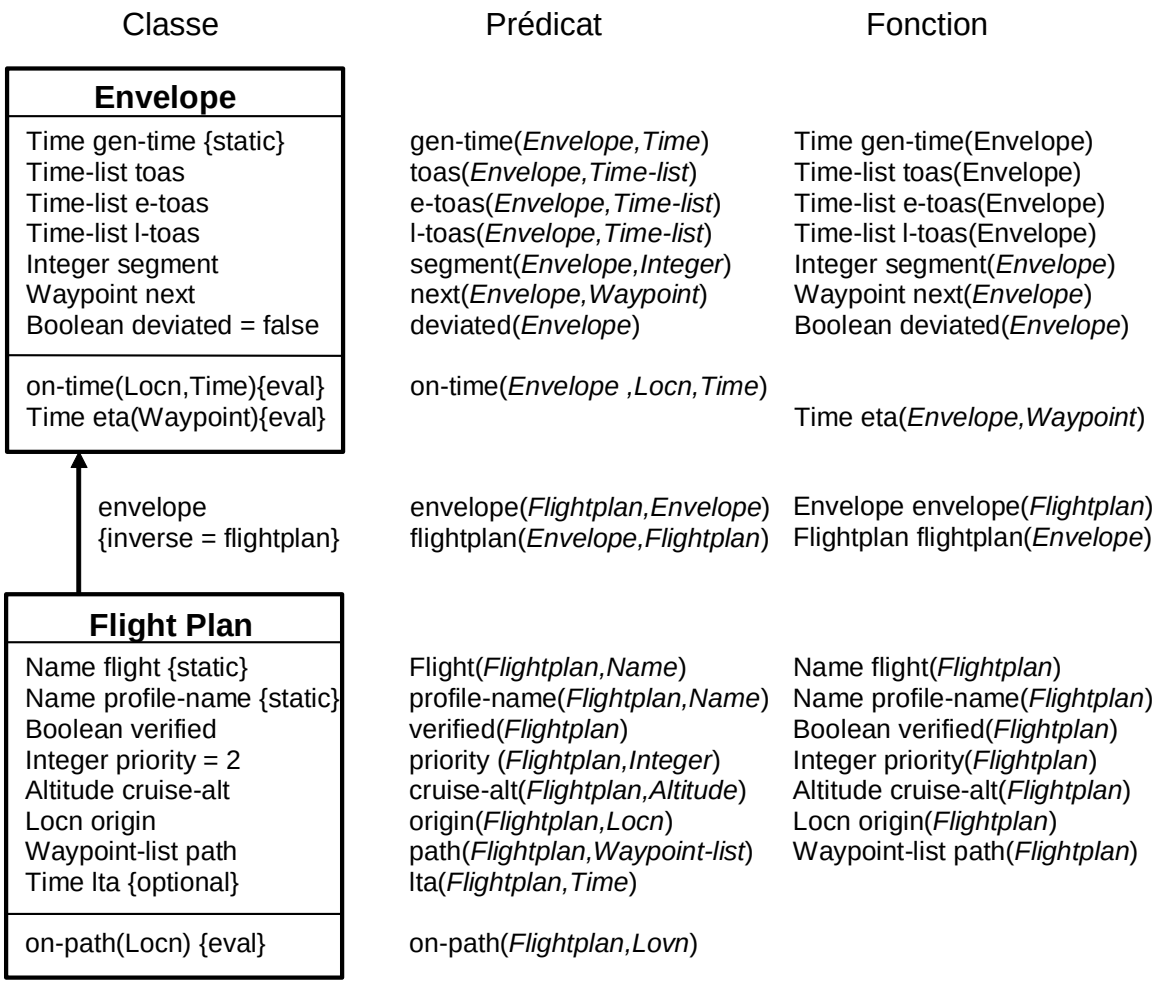


FIG. 2.2 – Exemple de classes de croyances.

Les classes de l'ensemble des croyances servent en fait à définir :
- les signatures de types des attributs des objets (ou "concepts") que cette classe englobe
- les fonctions qui peuvent être appliquées à ces objets
- les prédicats qui s'appliquent à ces objets, comme par exemple les *actions*, ayant un rôle central dans les *plans*

Chaque attribut d'une classe définit un prédicat (2ème colonne), et une fonction d'accès (3ème colonne), qu'il ne faut pas confondre avec les prédicats et fonctions *applicables* aux objets de la classe. Toutefois, ces derniers génèrent également des prédicats ou fonctions. L'objet auquel ils s'appliquent se trouve en premier argument. L'ensemble des prédicats dérivés constitue

l'*ensemble des croyances* à proprement parler.

Les attributs peuvent également être représentés par une association, quand ceux-ci désignent une autre classe. Comme pour les autres attributs, ceux-ci génèrent un prédicat liant cette fois ci les deux classes, et une fonction d'accès. Ce type d'association est représenté sur la figure 2.2 par la relation *envelope* et son inverse, *flightplan*.

Il est également possible d'étendre la notation des attributs et des opérations avec une liste de propriétés.

La similitude de ce modèle au modèle entité-association n'est pas fortuite ici. L'élaboration de ce modèle est finalement assez similaire à l'élaboration d'une base de données relationnelle, les relations étant concrétisées ici grâce à des prédicats de la logique du 1er ordre.

Pour finir, un *état de croyance* a une structure identique à l'*ensemble des croyances*. La différence réside dans le fait que les termes prennent des valeurs particulières. Ainsi, les états de croyances constituent généralement les états mentaux initiaux possibles de l'agent.

2.1.6 Le modèle de buts

Chaque classe d'agent est associée à un modèle de buts. Celui-ci est composé d'un *ensemble de buts* (goal-set) et d'un ou plusieurs *états de buts* (goal-states).

Un *ensemble de buts* est composé d'un ensemble de formules de buts, chacune composée d'un opérateur modal de but et d'un prédicat de l'*ensemble des croyances*. Les opérateurs modaux sont les suivants :

- achieve : représente le but d'accomplissement du prédicat associé. Ainsi, ce but est réussi si le prédicat associé est vrai (c'est-à-dire s'il se trouve dans l'ensemble des croyances), ou si l'exécution d'un plan associé rend le prédicat vrai. Il échoue si aucun plan ne rend le prédicat vrai.
- verify : représente le but de vérification du prédicat associé. Ce but est réussi si le prédicat associé est vrai, il est échoué si le prédicat est faux. Toutefois, si ce prédicat est indéterminé, un plan peut être déclenché, afin de rendre ce prédicat déterminé.
- test : représente le but de détermination du prédicat associé. Ce but est réussi si le prédicat associé est déterminé, c'est-à-dire s'il est vrai ou faux, mais pas inconnu. Dans le cas ou le prédicat est inconnu, un plan peut être déclenché. Dans ce cas-là, le but sera réussi si l'exécution du plan aboutit au fait que le prédicat n'est plus inconnu.

La différence entre l'*ensemble de buts* et un *état de but* est la suivante : une formule de l'*ensemble de buts* contient un prédicat provenant de l'*ensemble des croyances*, tandis qu'une formule d'un *état de but* contient un prédicat provenant d'un *état de croyance*.

2.1.7 Le modèle de plans

Un *modèle de plan* consiste en un ensemble de plans, chaque plan étant spécifié par un diagramme de transitions, à la manière d'un modèle dynamique OO. Ceux-ci expriment comment l'agent doit se comporter pour réagir à un événement.

La figure 2.3 présente le schéma générique d'un plan. Un plan comporte 3 types de noeuds : les noeuds de départ, les noeuds d'arrivée et les noeuds internes. Les noeuds internes peuvent être passifs ou actifs. Les noeuds passifs n'ont pas de sous structure, alors que les noeuds actifs sont associés à une formule d'activité. Cette activité peut être un but (qui déclenchera par exemple un autre plan), une construction itérative, ou un sous graphe. Les noeuds d'arrivée peuvent être soit des noeuds de réussite (◎) ou des noeuds d'échec (⌀).

Chaque transition est effectuée lorsque l'événement associé se produit. Ainsi le déclenchement d'un plan se fait lorsque l'événement de la première transition de ce plan se produit. Ces événements peuvent être de plusieurs natures. Premièrement, un événement peut être une croyance, qui est déclenché lorsqu'une certaine croyance est modifiée ou lorsqu'un un changement extérieur est perçu par l'agent. Deuxièmement, un événement peut être un but, généré par un autre plan par exemple.

Des conditions peuvent être associées aux transitions. Par conséquent, une transition a lieu lorsque l'événement à l'origine de la transition se produit, et que la condition est vraie. Cette condition est vraie lorsque le prédicat qu'elle contient appartient à l'*état de croyances* de l'agent.

Lorsque que la transition a eu lieu, une action est provoquée. Cette action peut être soit définie dans l'*ensemble des croyances* (en tant que prédicat applicable à une classe de croyance – cfr 2.1.5), soit une action prédéfinie, comme *assert()* et *retract()*, qui mettent à jour l'état de croyance de l'agent.

Nous pouvons voir que les modèles de croyances, de buts et de plans que nous venons de définir interagissent. En effet, un agent (tel que B737 dans la figure 2.1) disposant d'un état de croyances, d'un état de buts, ainsi que d'un ensemble de plans, déclenchera un ou plusieurs plans à partir de ses buts initiaux. L'exécution de ce plan créera éventuellement de nouveaux

buts (via les formules d'activité dans la figure 2.3), qui déclencheront d'autres plans. Cette exécution dépend aussi de l'état de croyance de l'agent à cause de certaines conditions associées aux transitions, et peut également modifier celui-ci.

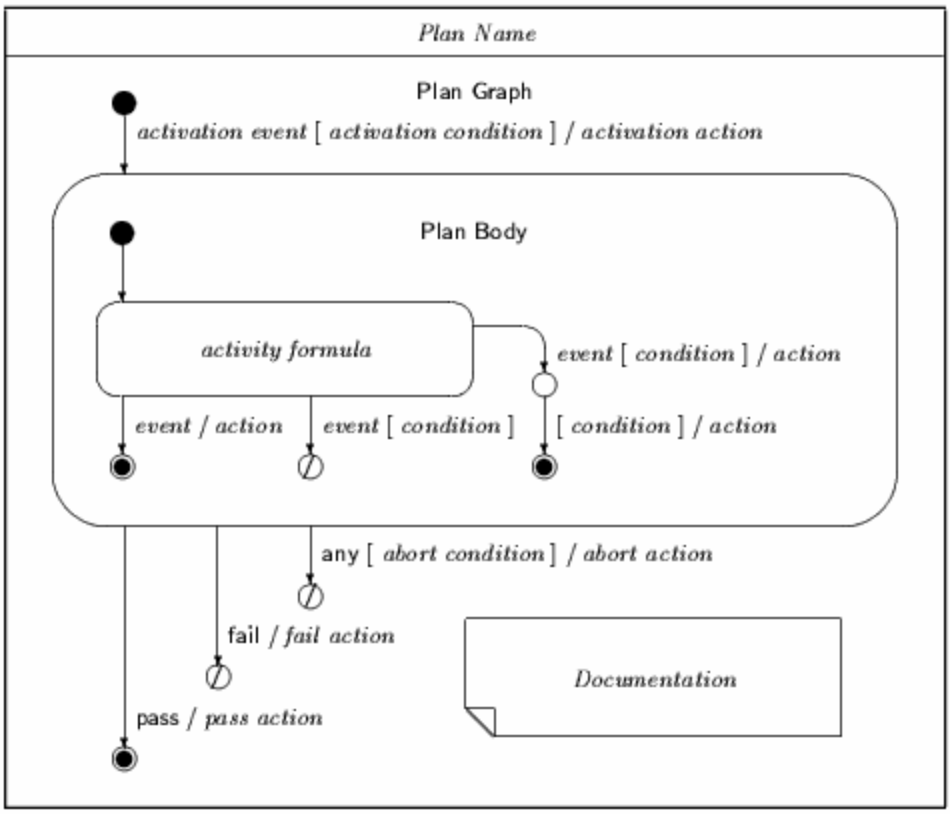


FIG. 2.3 – Schéma générique d'un plan.

2.1.8 Conclusion

Les modèles présentés ci-dessus fournissent une vue globale du système à implémenter, et ceux-ci peuvent être traduits dans un langage de programmation orienté-agent tel que 3APL. Le chapitre 8 fournit un exemple concret de construction d'un programme 3APL grâce à la méthodologie AAII.

La section suivante fournit présente quant à elle une autre méthodologie inspirée de l'orienté-objet : la méthodologie Gaia.

2.2 Méthodologie Gaia

M. Wooldridge, N. Jennings et D. Kinny proposent dans [51] la méthodologie Gaia, qui permet de dériver systématiquement un ensemble de besoins en un ensemble de modèles qui puissent être implémentés facilement. Tout comme la méthodologie AAII, Gaia s'inspire directement de la notation orientée-objet, tout en étendant celle-ci pour capturer les concepts propres aux agents.

Cette méthodologie se veut plus générale que la méthodologie AAII, étant donnée qu'elle est indépendante du formalisme particulier BDI. Toutefois, les auteurs ont jugé bon de restreindre le domaine d'application de cette méthodologie en émettant un certain nombre d'hypothèses [51] :

- Les agents sont des entités complexes (*coarse-grained)*, utilisant chacune des ressources significatives (on peut voir un agent comme un processus UNIX)
- Le but est d'obtenir un système qui maximise une qualité globale, mais qui peut ne pas être optimal du point de vue de ses sous-systèmes
- Les agents sont hétérogènes, en ce sens qu'ils peuvent être implémentés en utilisant des langages de programmation, des architectures et des techniques différents.
- La structure du système est statique, c'est-à-dire que les relations entre agents ne changent pas à l'exécution.
- Les capacités des agents ainsi que les services qu'ils fournissent sont statiques.
- Le système ne contient qu'un nombre relativement faible de types d'agents (moins de 100 en général)

La méthodologie Gaia est composée de deux étapes : l'*analyse* et le *design*. L'application successive de ces deux étapes permet d'obtenir des modèles de plus en plus détaillés. La figure 2.4 expose les différents modèles utilisés, ainsi que leurs relations.

Nous détaillerons maintenant les différents concepts propres à l'étape d'analyse, et la constitution de ceux-ci en modèles.

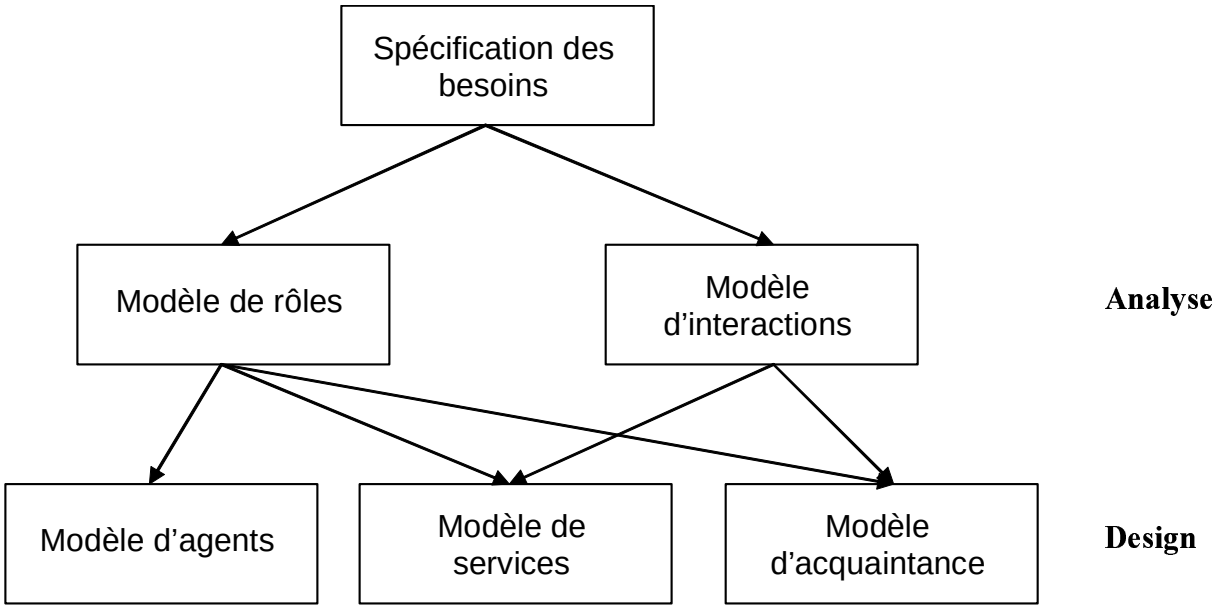


FIG. 2.4 – Modèles utilisés dans la méthodologie Gaia.

2.2.1 Les modèles d'analyse

Concepts utilisés

Tout comme dans la méthodologie AAII, le système est vu comme un ensemble de *rôles* ayant un certain nombre d'*interactions*. Un rôle est défini par quatre attributs : ses *responsabilités*, ses *permissions*, ses *activités* et les *protocoles* qu'il entretient avec d'autres rôles.

Les *responsabilités* sont certainement l'attribut principal d'un rôle, car elles définissent les fonctionnalités de celui-ci. Les responsabilités sont séparées en deux classes : les "*propriétés de vivacité*" et les "*propriétés de sécurités*". Intuitivement, les premières définissent ce qu'un rôle est censé faire, sa raison d'être en quelque sorte. Les deuxièmes définissent quant à elles les invariants que doit respecter un rôle.

Les *permissions* permettent à un rôle de mettre en oeuvre ses responsabilités. Elles identifient les ressources auxquelles a accès un rôle.

Les *activités* et les *protocoles* constituent les actions de base, qui se combinent entre elles pour constituer les responsabilités. Les *activités* sont des actions privées, en ce sens qu'elles sont effectuées par un rôle sans interagir avec d'autres rôles. Les *protocoles* définissent comment un rôle interagit avec d'autres rôles.

Le modèle de rôles

Comme son nom l'indique, le modèle de rôles définit l'ensemble des rôles présents dans le système. Dans ce modèle, chacun des rôles est défini par ses *permissions* et ses *responsabilités*. Notons que les activités et protocoles ne sont pas définis dans ce modèle, ils sont simplement utilisés en vue de définir les responsabilités.

Les **permissions** consistent en les ressources qu'un rôle peut utiliser, ainsi que les limites imposées au rôle dans l'utilisation de ces ressources. Dans la méthodologie Gaia, les ressources se limitent seulement à l'information, toutefois, les auteurs [51] reconnaissent que cette notion de ressource devrait être étendue par la suite.

La notation utilisée pour exprimer les permissions est inspirée de celle utilisée dans FUSION [7]. Trois types de permissions existent : les permissions *read*, *change* et *generate*. La première indique qu'un rôle a le droit d'accéder à une information, la deuxième indique qu'un rôle a aussi bien le droit d'accéder que de modifier une information, la dernière indique que le rôle est un producteur d'une ressource.

Pour illustrer, reprenons l'exemple fourni dans [51], définissant les permissions du rôle *CoffeeFiller*, dont le but est de s'assurer qu'un récipient est continuellement rempli de café pour un groupe de travailleurs. La définition de ses permissions est la suivante :

```
Reads       coffeeStatus     // plein ou vide
Changes     coffeeStock      // niveau de stock de café
```

Ce rôle a donc accès à la valeur *coffeeStatus* et a également le droit de modifier la valeur *coffeeStock*. Il est toutefois plus intéressant de paramétrer ce rôle avec la machine à café qu'il est censé remplir. La définition devient donc la suivante :

```
Reads   supplied coffeeMaker   // nom de la machine à café
        coffeeStatus           // plein ou vide
Changes coffeeStock            // niveau de stock de café
```

Les **responsabilités**, comme nous l'avons dit précédemment, sont divisées en deux catégories : les *propriétés de vivacité* et les *propriétés de sécurité*.

Les *propriétés de vivacité* garantissent qu'un agent est toujours "en vie", en spécifiant par exemple qu'une réponse est émise pour chaque requête. Idéalement, les propriétés de vivacité devraient être formalisées grâce à la logique modale (dont discutera le chapitre 4), mais par souci de simplicité, elles sont définies par des expressions régulières, augmentées de l'opérateur de répétition infinie (∞). Les composants atomiques de ces expressions sont soit des activités, soit des protocoles. Les opérateurs utilisés par ces expressions régulières sont définis dans la figure 2.5.

Opérateur	Interprétation
x , y	*x suivi de y*
x | y	*x ou y surviennent*
*x**	*x survient 0 ou plusieurs fois*
x+	*x survient 1 ou plusieurs fois*
x^{∞}	*x survient une infinité de fois*
[*x*]	*x est optionnel*
x || y	*x et y simultanément*

FIG. 2.5 – Opérateurs des expressions de vivacité.

Reprenons l'exemple du *CoffeeFiller*. Sa *propriété de vivacité* consiste à remplir la machine à café, à en informer les travailleurs, à vérifier le stock de café et finalement d'attendre que la machine à café soit à nouveau vide, et tout ceci dans une boucle infinie. Ceci est exprimé de la manière suivante :

$$\text{CoffeeFiller} = (\text{Fill.InformWorkers.}\underline{\text{CheckStock}}\text{.AwaitEmpty})^{\infty}$$

Notons que les activités sont distinguées des protocoles en étant soulignées.

Les *propriété de sécurité* sont les invariants que doit respecter un rôle, et sont représentées par une liste de prédicats, qui sont généralement exprimés sur les variables définies dans les permissions. Dans notre exemple du *CoffeeFiller*, il faut qu'à tout moment que le stock de café soit positif, ce qui exprimé comme suit :

```
coffeeStock > 0
```

Nous pouvons donc maintenant définir ce qu'est le *modèle de rôles*. Un modèle de rôles est composé d'un ensemble de schémas de rôles. La figure 2.6 représente un schéma de rôle générique. La figure 2.7 quant à elle est le schéma de rôle de l'exemple du *CoffeeFiller*.

Schéma de rôle :	*Nom du rôle*
Description	*description informelle du rôle*
Protocoles et Activités	*Protocoles et activités dans lesquels le rôle intervient*
Permissions	*Droits associés au rôle*
Responsabilités	
Liveness	*Liveness properties du rôle*
Safety	*Safety properties du rôle*

FIG. 2.6 – Schéma de rôle générique.

Schéma de rôle : COFFEEFILLER

Description :
Ce rôle consiste à assurer qu'un récipient est toujours rempli de café, et à informer les travailleurs quand du café frais a été fait

Protocoles et Activités :
`Fill, InformWorkers, CheckStock, AwaitEmpty`

Permissions :

reads	**supplied** *coffeeMaker*	*// nom de la machine à café*
	coffeeStatus	*// plein ou vide*
changes	*coffeeStock*	*// niveau de stock de café*

Responsabilités
Liveness :
COFFEEFILLER = (`Fill.InformWorkers.CheckStock.AwaitEmpty`)$^{\omega}$

Safety :
coffeeStock > 0

FIG. 2.7 – Schéma de rôle de CoffeeFiller.

Le modèle d'interactions

Le modèle d'interactions définit l'ensemble des *protocoles* liant les différents rôles du système, ces mêmes protocoles qui sont utilisés pour la définition des *responsabilités* des rôles dans le *modèle de rôles*. Le but à ce niveau ci n'est pas de définir l'ensemble des messages échangés entre rôles de manière exhaustive, car nous n'en sommes toujours qu'à la phase d'analyse. La définition d'un protocole consiste donc à définir un ensemble d'attributs :

- le *but*, qui consiste à décrire de manière textuelle la nature de l'interaction
- l'*initiateur*, qui est le(s) rôle(s) responsable(s) du démarrage de l'interaction
- le *répondeur*, qui est le(s) rôle(s) avec qui l(es) initiateur(s) interagit
- les *entrées*, qui sont les informations utilisées par l'initiateur en vue d'établir l'interaction
- les *sorties*, qui sont les informations fournies par le répondeur ou à celui-ci
- les *processings*, qui consistent à définir les différentes procédures que l'initiateur effectue pendant l'interaction

Ainsi, dans le cas de l'exemple du *CoffeeFiller*, le protocole Fill est défini de la manière suivante :

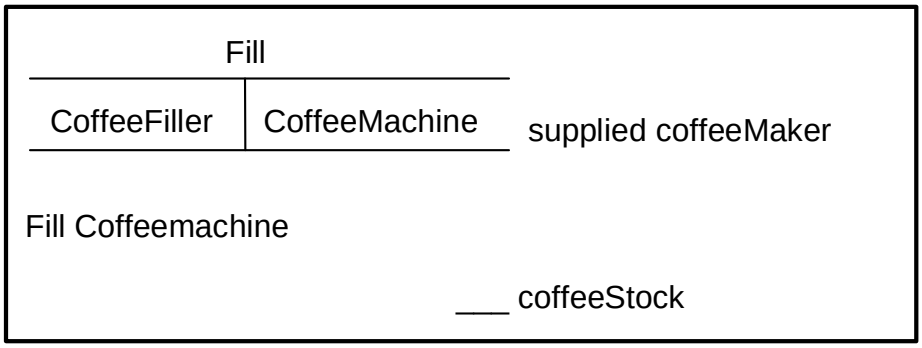


FIG. 2.8 – Protocole Fill.

Ceci définit que le protocole Fill est initié par le rôle *CoffeeFiller* et implique le rôle *CoffeeMachine*. Durant cette interaction, le rôle *CoffeeFiller* est amené à remplir la machine à café (*Fill coffee machine*). Le rôle *CoffeeMachine* est informé quant à lui sur la valeur de *coffeeStock*.

Maintenant que nous connaissons les modèles de la phase d'analyse, nous pouvons examiner comment élaborer ces modèles.

2.2.2 Le processus d'analyse

- Identifier les différents rôles présents dans le système, ce qui débouche sur un *modèle de rôles* primitif, avec une description informelle pour chacun de ces rôles
- Pour chacun des rôles, identifier les *protocoles* associés, ainsi que ses activités. A ce stade-ci, on dispose d'un *modèle d'interactions*
- A partir du *modèle d'interactions*, établir le *modèle de rôles* complet, avec pour chacun des rôles ses permissions et ses responsabilités

Maintenant que les modèles de l'étape d'analyse sont constitués, nous pouvons étudier les modèles de design.

2.2.3 Les modèles de design

Le processus de design consiste à élaborer trois modèles : le modèle d'agents, le modèle de services et le modèle de relations.

Le modèle d'agents

Le but du modèle d'agents est de définir les *types d'agents* du système, ainsi que les *instances d'agents* qui seront présentes à l'exécution. Le modèle d'agent consiste donc en un arbre de types d'agents. Chaque type d'agent peut représenter un certain nombre de rôles, le choix de regroupement de ces rôles étant guidé généralement par des critères d'efficacité. Les relations entre types d'agents dans l'arbre sont des relations d'inclusion (et non des relations d'héritage). Ainsi, les feuilles de l'arbre correspondent à des rôles, tels que définis dans le *modèle de rôles*. Les noeuds consistent en des *types d'agents* qui sont des agrégations des rôles de leurs noeuds-fils. Les *instances d'agents* présentes à l'exécution sont représentées par une annotation des *types d'agents* par une cardinalité, indiquant ainsi le nombre d'agents de chaque type présents à l'exécution.

Le modèle de services

Le modèle de service fournit pour chacun des services associés à un agent les entrées, sorties, pré-conditions et post-conditions de ce service. L'ensemble des services est dérivé des *protocoles*, *activités* et *responsabilités* définis pendant l'étape d'analyse. Ainsi, par exemple, à chacun des protocoles correspondra un service. Les entrées et sorties de ce service seront évidemment dérivés de la définition du protocole dans le *modèle d'interactions*. Les pré et post-conditions seront dérivées quant à elles des *propriétés de sécurité* du rôle. Il faut toutefois préciser que la relation liant les activités et protocoles aux services n'est pas forcement unaire. On pourrait ainsi envisager de découper un protocole en un ensemble de services, par souci d'efficacité.

Le modèle de relations

Ce modèle sert simplement à décrire les liens entre les différents types d'agents. Il s'agit donc d'un graphe orienté dont les noeuds sont les *types d'agents* dégagés dans le *modèle d'agents*, et dont les arcs indiquent que l'origine envoie des messages au destinataire. Notons que l'on ne s'intéresse pas ici aux messages échangés, mais bien aux canaux de communication.

2.2.4 Le processus de design

- Créer un *modèle d'agents*, en agrégeant les différents rôles du *modèle de rôles* pour former une hiérarchie de *types d'agents*. Indiquer également le nombre d'instances de chacun de ces types d'agents qui seront présentes à l'exécution
- Créer le *modèle de services* à partir des différents *protocoles* et *activités*, ainsi que des *propriétés de vivacité* et des *propriétés de sécurité*
- Créer le *modèle de relations*

2.3 Evaluation

Les deux méthodologies qui viennent d'être exposés sont certainement les plus répandues. Bien qu'elles s'inspirent toutes deux du développement orienté-objet, elles présentent des différences importantes.

Premièrement, la méthodologie Gaia est plus générale que la méthodologie AAII, en ce sens qu'elle est totalement indépendante du type d'architecture des agents du système final. La méthodologie AAII impose quant à elle l'architecture BDI. Ceci a aussi bien des avantages que des inconvénients : la méthodologie AAII, en fournissant des modèles de croyances, de plans et de buts, permet d'implémenter de manière relativement aisée le système grâce à un langage de programmation agent, tel que 3APL. Malheureusement, il paraît non approprié d'utiliser la méthodologie AAII pour implémenter des agents purement réactifs par exemple. La méthodologie Gaia, quant à elle, reste à un niveau d'abstraction plus élevé, en ne fournissant que des modèles d'agents, de services et de relations. Il semble toutefois qu'à partir de ces modèles, beaucoup de travail reste à faire pour implémenter le système. Particulièrement si le développeur décide d'utiliser l'architecture BDI, qui est malgré tout l'architecture la plus répandue. La méthodologie Gaia ne fournit en effet aucun indice sur la manière d'implémenter les croyances, désirs et intentions de l'agent.

Deuxièmement, la méthodologie Gaia semble plus systématique que la méthodologie AAII, et par là même plus facile à appliquer. En effet, la méthodologie Gaia est clairement séparée en deux phases successives : l'analyse et le design. Bien que la méthodologie AAII ait distingué le modèle interne du modèle externe, la conception de ces derniers n'est pas nettement séparée, ce qui ne joue pas en la faveur de la clarté de la méthodologie.

Finalement, le choix entre ces deux méthodologies doit se faire au cas par cas. Si l'on a l'intention d'implémenter un système BDI, la méthodologie AAII parait s'imposer. Par contre, si le système multi-agent que l'on veut implémenter est principalement basé sur l'interaction des agents, plutôt que sur leur "comportement intelligent", notre choix se portera plutôt sur la méthodologie Gaia.

Comme il l'a été dit précédemment, ces méthodologies s'inspirent principalement de la conception orientée-objet, et ceci n'est pas sans poser certains problèmes. Comme l'indique M. Wooldridge [52], bien que les concepts orientés-objet soient de plus en plus familiers à une grande audience de développeurs, la nature de la décomposition propre à la conception orientée-objet n'est pas la même que celle qu'encourage la conception orientée-agent. De plus, les méthodologies orientées-objet ne sont pas à même de modéliser des aspects importants des systèmes multi-agents, tels que la pro-activité, la négociation, la coopération, etc., bien qu'il y ait eu des efforts accomplis pour étendre UML à ces aspects [1].

On peut d'ailleurs se demander s'il est pertinent de se baser sur le paradigme orienté-objet pour "créer" le paradigme orienté-agent. En effet, si ce dernier mérite le qualificatif de "paradigme", il est censé fournir un nouveau point de vue sur la conception logicielle que ne peut appréhender le paradigme orienté-objet. Le chapitre suivant présentera donc une autre méthodologie, se basant sur l'ingénierie des connaissances.

Chapitre 3

Méthodologies inspirées de l'ingénierie des connaissances

L'*ingénierie des connaissances*, ou *knowledge engineering* (KE) [47], est une discipline dont le but est de développer des systèmes basés sur la connaissances, ou *knowledge based systems* (KBS). Ces systèmes consistent généralement à automatiser des processus de décision complexes, tel que le diagnostic médical. Le développement de ces systèmes a évolué au cours des vingt dernières années de la vision d'un *processus de transfert* des connaissances humaines vers la vision d'un *processus de modélisation* [47]. Le but n'est plus de simuler les compétences cognitives d'un expert dans un domaine particulier, mais bien de créer un modèle qui puisse résoudre des problèmes dans ce domaine particulier. Le processus d'acquisition des connaissances est donc passé d'un processus de transfert de connaissances à un processus de construction de modèle. Par conséquent, plusieurs méthodologies de conception de systèmes basés sur la connaissance ont été proposées, dont la méthodologie CommonKADS [44], qui a été étendue pour la conception de systèmes multi-agent par la méthodologie MAS-CommonKADS [29, 26].

Le point 1 exposera les fondements de la méthodologie CommonKADS (et en particulier le modèle d'expertise). Le point 2 exposera les évolutions apportées par la méthodologie MAS-CommonKADS afin de rendre compte des spécificités des systèmes multi-agents. Mais avant toute chose, il est important d'introduire certains concepts propres à l'ingénierie des connaissances :

- L'étude des systèmes experts de première génération a permis de dégager une méthode commune de résolution des problèmes appelée *Heuristic Classification* [5], ce qui a débouché sur le concept de **problem-solving method** (PSM). Une PSM constitue une méthode générique de résolution de problèmes. Par la suite, d'autres PSM ont été identi-

fiées, telles que *Cover-and-Differentiate* et *Propose-and-Revise*. La figure 3.1 illustre la PSM *Heuristic Classification*.

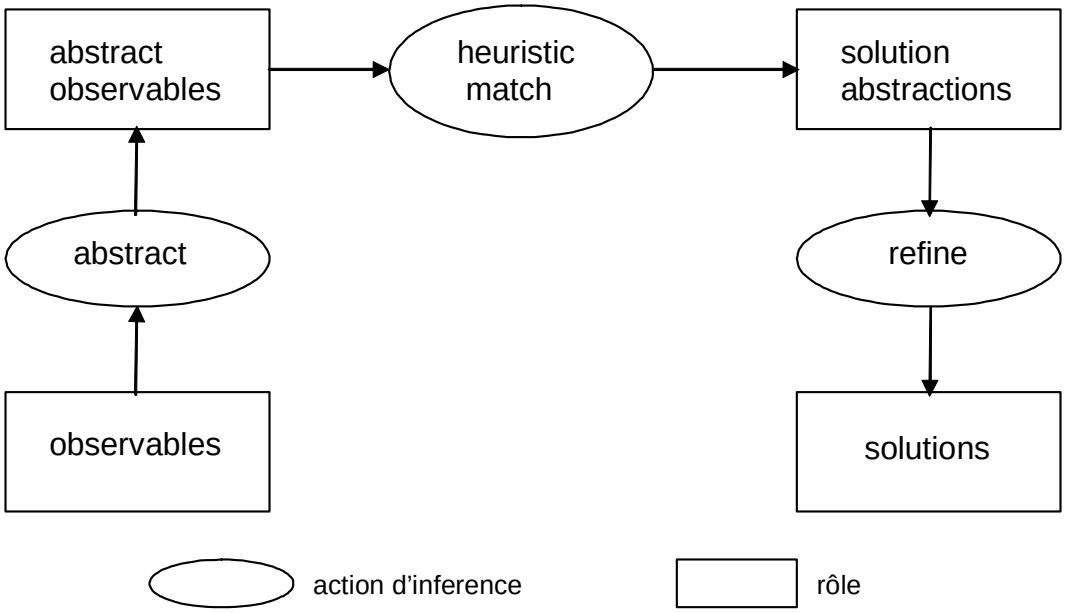


FIG. 3.1 – PSM *Heuristic Classification*.

Dans la figure 3.1, nous pouvons observer trois actions d'inférence : *abstract*, *inference match* et *refine*. De plus, quatre *rôles de connaissances* sont distingués : *observables*, *abstract observables*, *solution abstractions* et *solutions*. Ces rôles présentent l'avantage d'être génériques, et peuvent être instanciés par des "valeurs" d'un domaine d'application particulier.

La réutilisabilité de ces PSM a permis donc de supporter le développement des KBS. Les PSM fournissent en effet des rôles génériques, le développeur n'ayant plus qu'à définir les concepts du domaine d'application auxquels les rôles correspondent, ainsi que des instances pour chacun de ces concepts. Par exemple, dans le domaine du diagnostic médical, le rôle *observables* de la PSM *Heuristic Classification* correspondra aux données relatives aux patients, dont une instance pourrait être '*température de 40° C*'.

– Une deuxième notion importante de l'ingénierie des connaissances est celle de *Tâche Générique* ou **Generic Task** (GT). Les GT permettent de réutiliser certaines routines dans différents KBS. Une GT consiste en une description des ses entrées et sorties, et de la séquence des étapes

d'inférence de la PSM associée à cette GT. Ainsi, la notion de tâche se trouve à un niveau d'abstraction supérieur à celle de PSM, les PSM servant à accomplir une tâche.

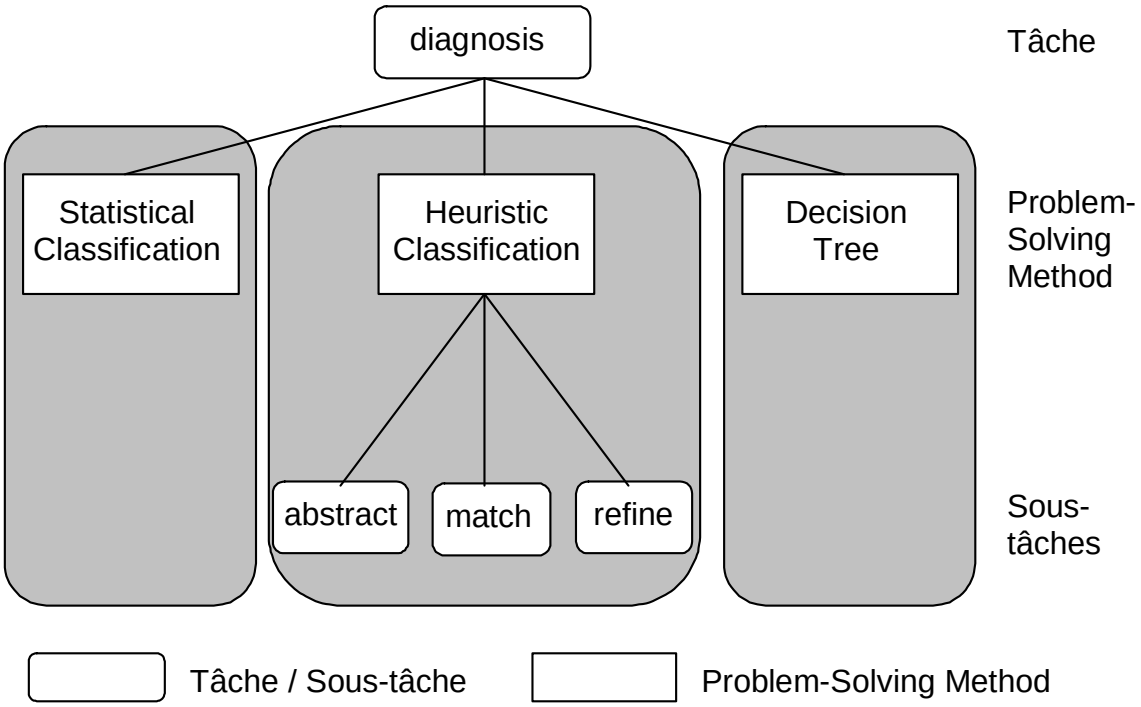


FIG. 3.2 – Exemple de *Task Structure*.

Un inconvénient des *Generic Tasks* est qu'elles sont fournies avec une PSM prédéterminée, ce qui peut mener à confondre la notion de tâche et de PSM. Pour pallier à ceci, la notion de *Task Structure* a été proposée [4], dont nous pouvons voir un exemple à la figure 3.2. Nous pouvons y observer une relation claire entre tâche et PSM, les tâches pouvant être résolues par différentes PSM, chacune de ces PSM utilisant à son tour plusieurs sous-tâches. Ces sous-tâches peuvent à leur tour être décomposées, sauf s'il s'agit d'actions d'inférence élémentaires.

Connaissant ces concepts, nous pouvons maintenant définir les fondements principaux de la méthodologie CommonKADS.

3.1 Méthodologie CommonKADS

Avant toute chose, il est important de préciser que la méthodologie CommonKADS n'est pas une méthodologie orientée-agent, il s'agit d'une méthodologie permettant la conception de systèmes basés sur la connaissance, sur laquelle se base la méthodologie MAS-CommonKADS.

La méthodologie CommonKADS définit plusieurs modèles : le *modèle d'organisation*, le *modèle de tâche*, le *modèle d'agent*, le *modèle de communication*, le *modèle d'expertise* et le *modèle de design*. L'atout principal de la méthodologie CommonKADS réside dans la structure proposée pour le *modèle d'expertise*, c'est pourquoi nous ne détaillerons que ce modèle dans ce point-ci. Les autres modèles seront détaillés dont la description de la méthodologie MAS-CommonKADS.

Le modèle d'expertise distingue trois couches de connaissances : la couche du *domaine*, la couche d'*inférence*, la couche de la *tâche*, correspondant respectivement à une vue statique, fonctionnelle et dynamique du système à construire. La figure 3.3 illustre un exemple de modèle d'expertise pour le diagnostic médical.

- La *couche de la tâche* (task layer) fournit une décomposition des tâches en sous-tâches et actions d'inférence. Elle spécifie également pour chaque tâche son but, ses entrées et ses sorties.
- Comme nous l'avons dit précédemment quand nous avions introduit la notion de *Task Structure*, une tâche est mise en oeuvre par une PSM. La *couche d'inférence* (inference layer) définit cette PSM, spécifiant les actions d'inférences ainsi que les rôles qui sont joués par les connaissances du domaine.
- Les connaissances du domaine sont modélisées dans la *couche du domaine* (domain layer), sous la forme d'ontologies [47]. Comme nous pouvons le voir sur la figure 3.3, chacune des connaissances spécifiques au domaine modélisées dans la couche du domaine correspond à un rôle de la couche d'inférence.

Comme le précise [47], la modularité de ce modèle permet deux types de réutilisabilité :

- une description de la couche du domaine peut être réutilisée pour résoudre d'autres tâches avec d'autres PSM
- une PSM donnée peut être réutilisée pour un autre domaine

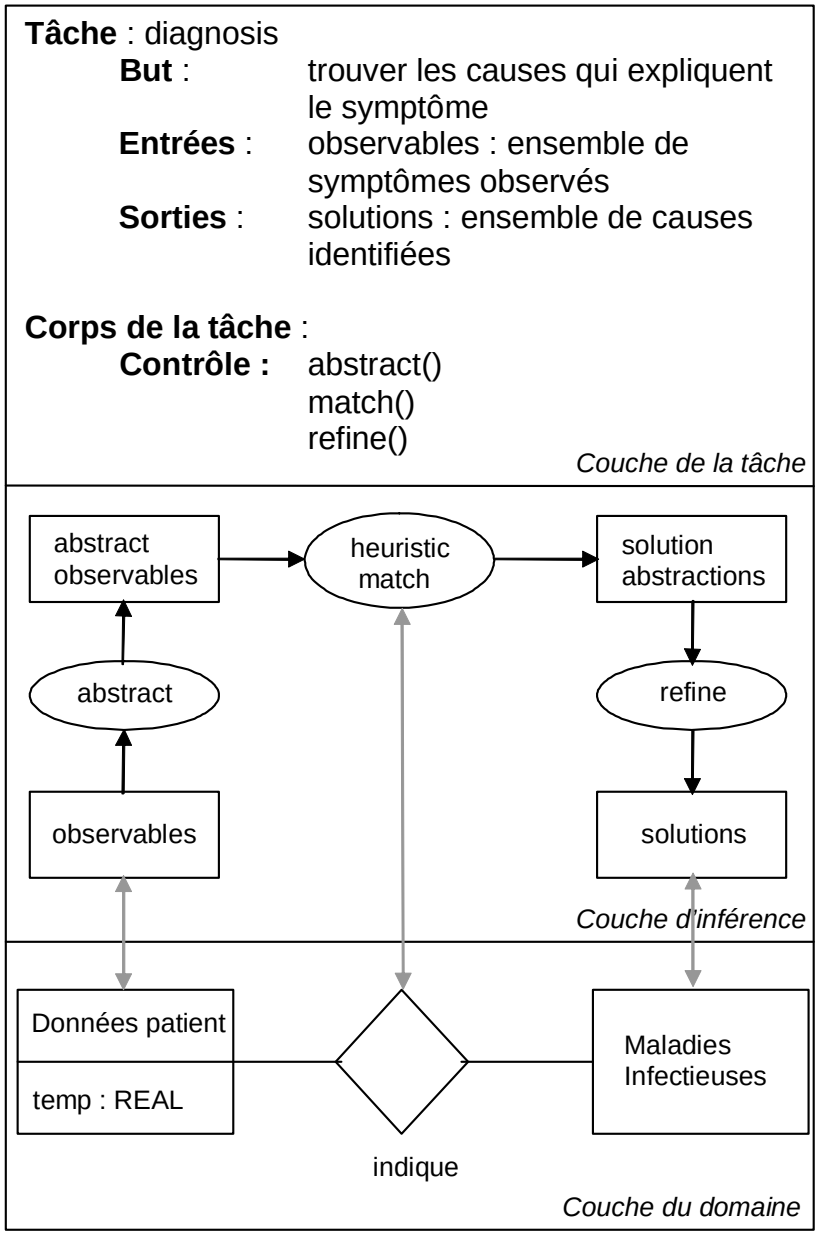


FIG. 3.3 – Exemple de modèle d'expertise pour le diagnostic médical.

La méthodologie MAS-CommonKADS reprend ce modèle d'expertise, tout en apportant certaines modifications aux autres modèles, que nous décrirons dans le point suivant.

3.2 Méthodologie MAS-CommonKADS

La méthodologie MAS-CommonKADS [29, 26] étend la méthodologie CommonKADS en y ajoutant des techniques provenant des méthodologies OO (OMT [43], OOSE [31]), et de l'ingénierie des protocoles (SDL [30], MSC [42]). La méthodologie MAS-CommonKADS définit 7 modèles (le *modèle d'agent*, le *modèle de tâche*, le *modèle d'expertise*, le *modèle d'organisation*, le *modèle de coordination*, le *modèle de communication* et le *modèle de design*) au travers de 3 phases : la conceptualisation, l'analyse et le design. Par rapport à CommonKADS, on remarquera que le modèle de coordination est ajouté, l'apport principal de MAS-CommonKADS résidant justement dans ce modèle. Le modèle de communication était en effet limité aux interactions homme-machine, et ne pouvait donc pas rendre compte des communications entre agents.

3.2.1 Conceptualisation

La phase de conceptualisation consiste à obtenir une première description du problème, grâce à l'utilisation de *use cases*. Les *use cases* sont formalisés grâce à la notation OOSE et les interactions grâce à MSC. La figure 3.4 fournit un exemple de *use case* dans le cas d'une agence de voyage [29]. La figure 3.5 fournit un exemple d'interaction entre un utilisateur et le système modélisée grâce à MSC. Il est important de noter que la notation OOSE est étendue aux "utilisateurs agent", représentés par une tête carrée, et que l'opérateur *alt* représente une alternative entre plusieurs échanges de messages.

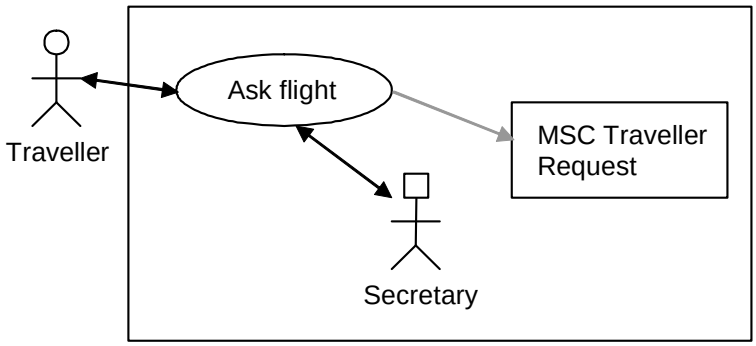


FIG. 3.4 – Use case.

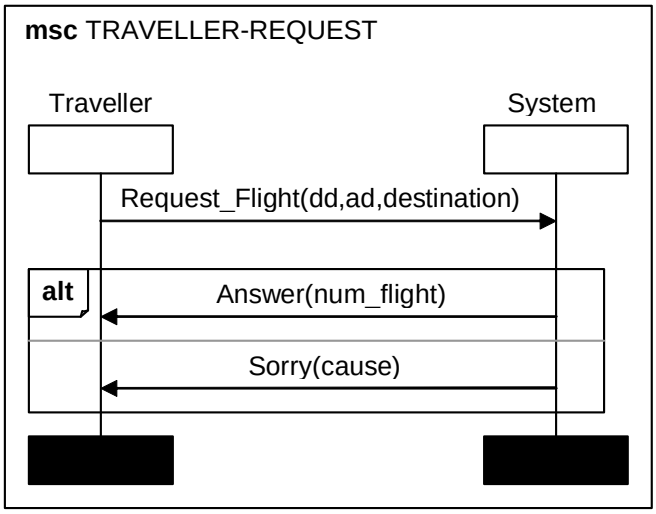


FIG. 3.5 – Interactions modélisées avec MSC.

3.2.2 Analyse

Le modèle d'agents

Un agent a cinq attributs : le *nom*, le *type* (humain, nouvel agent ou agent prédéfini), sa *superclasse*, son *rôle*, sa *position*, et les *groupes* auxquels il appartient. Il est également caractérisé par :

- les *services* qu'il propose aux autres agents. Un service a quatre attributs : son *nom*, son *type*, la *tâche* qu'il accomplit (provenant du modèle de tâches), et ses *ingrédients*
- ses *buts*, qui possèdent les attributs suivants : son *nom*, sa *description*, son *type* et ses *ingrédients*
- ses *capacités de raisonnement*, définies en faisant référence à une ou plusieurs instances du *modèle d'expertise* (cfr. infra). Les capacités de raisonnement de l'agent permettent d'accomplir ses buts
- ses *capacités générales*, consistant en ses aptitudes (senseurs et effecteurs) et en les langages de communication inter-agent et de représentation des connaissances qu'il comprend
- ses *contraintes*, qui sont ses normes, ses préférences et ses permissions. Elles aussi sont définies en faisant référence à une ou plusieurs instances du *modèle d'expertise* (cfr. infra), permettant par exemple de modéliser quand un agent décide de négocier

L'identification des différents agents peut se faire grâce à plusieurs méthodes non exclusives [29] :

- analyser les acteurs des *use cases* définis lors de la conceptualisation. Par exemple, l'existence d'un acteur humain débouchera sur la création d'un agent chargé de l'interface avec l'utilisateur
- analyser la syntaxe de l'énoncé textuel du problème, en affectant par exemple un agent pour les sujets des phrases.
- utiliser certaines heuristiques pour identifier de nouveaux agents, telles que : la distribution géographique (un agent pour chaque position physique), la distribution logique (le regroupement de certaines tâches élaborées dans le *modèle de tâches* en un but permet de créer un agent), la distribution des connaissances (créer des agents experts dans un domaine du *modèle d'expertise*)

Ainsi, dans l'exemple de l'agence de voyage évoqué précédemment, trois agents sont identifiés : un agent *Secretary*, chargé de l'interface avec l'utilisateur ; un agent *Predictor* qui est expert dans la prévision des vols ; un agent *Airlines-Clerck*, dont les instances sont dispersées géographiquement.

Le modèle de coordination

L'élaboration du modèle de coordination se fait en deux étapes. Premièrement, il s'agit de définir les canaux de communication et de créer un prototype du modèle. Deuxièmement, il s'agit d'analyser les interactions et de déterminer les interactions complexes.

La première étape consiste à :

- décrire les scénarios prototypiques entre agents en utilisant la notation MSC (figure 3.6). A cette étape-ci, une conversation consiste en une simple interaction
- représenter les messages échangés entre agents dans un diagramme de flux de messages (figure 3.7)
- modéliser les données échangées dans chaque interaction, représentées entre crochets dans le diagramme de flux de messages (figure 3.7)
- modéliser le *processus* chaque interaction grâce à un diagramme état-transition SDL (figure 3.8)

La deuxième étape consiste premièrement à optimiser les interactions dégagées à la première étape, en profitant par exemple du parallélisme, en

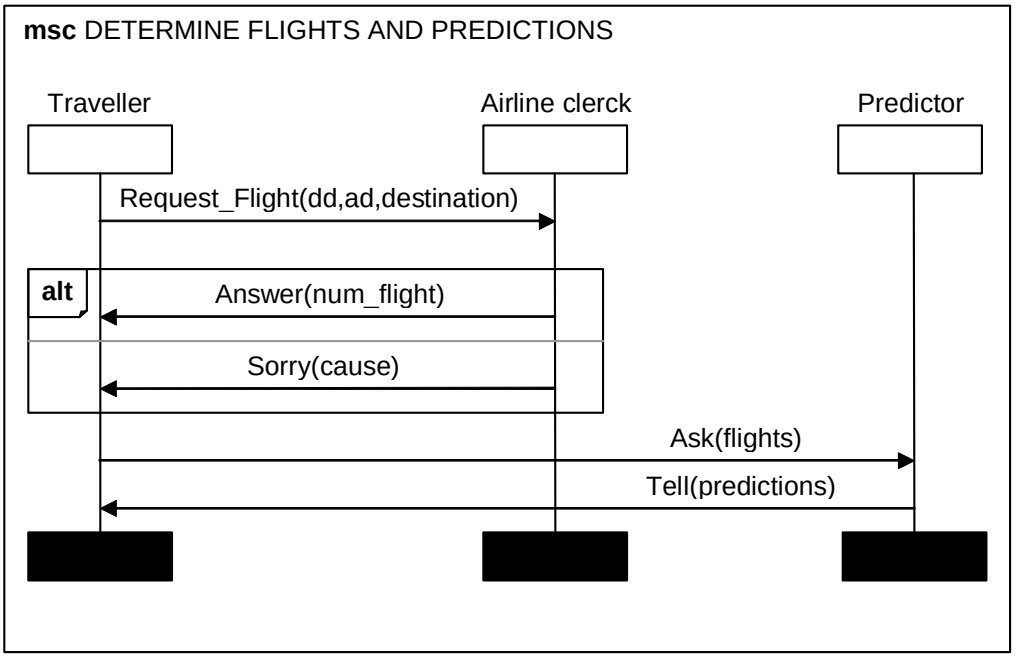


FIG. 3.6 – Modélisation des conversations avec MSC.

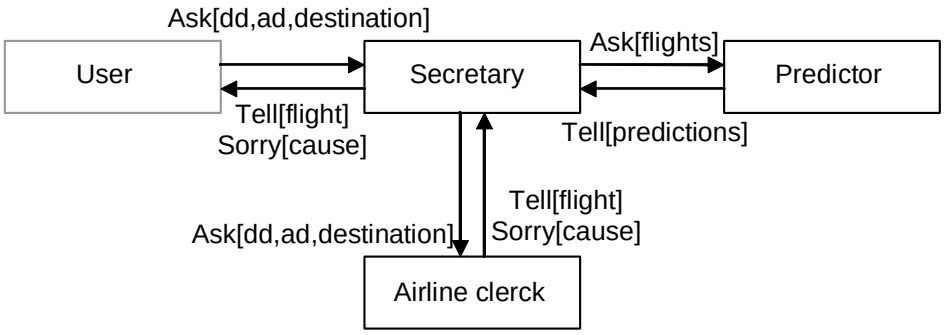


FIG. 3.7 – Diagramme de flux de messages.

résolvant les conflits, etc. Ensuite, dans le cas où un protocole de coopération est requis, il s'agit de soit consulter la bibliothèque des protocoles de coopération déjà créés par le passé, soit, le cas échéant, en créer un nouveau. Dans ce dernier cas, les auteurs [29] proposent d'utiliser les HMSC (High Level Message Sequence Charts) [30]. Ce formalisme, dont un exemple est présenté à la figure 3.9, permet de définir des enchaînements d'interactions simples (définies grâce aux MSC) et complexes (définies grâce aux HMSC).

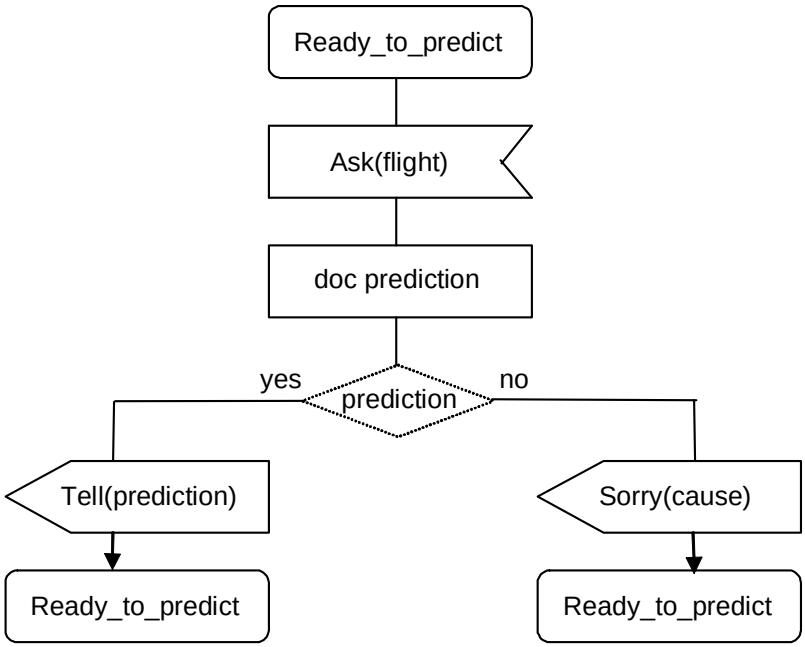


FIG. 3.8 – Diagramme état-transition SDL.

A l'instar des interactions modélisées avec les MSC, le *processus* des protocoles de coopération définis avec les HMSC peut être décrit grâce à un diagramme état-transition SDL. Il est également nécessaire de définir les *capacités de raisonnement* des agents nécessaires au déroulement du protocole, encore une fois en faisant référence à une ou plusieurs instances du *modèle d'expertise*.

Le modèle d'organisation

Alors que le *modèle de coordination* définit les relations dynamiques entre agents, le *modèle d'organisation* définit les relations statiques et structurelles. Ce modèle est défini grâce au modèle objet de OMT, en y ajoutant un symbole spécial permettant de distinguer les agents des objets. La figure 3.10 présente un exemple de modèle d'organisation.

La partie supérieure de l'entité représentant un agent contient les attributs internes à l'agent et son état mental (tel que son but). La partie inférieure contient les attributs externes, c'est-à-dire ses services, senseurs et effecteurs. La relation d'héritage reste inchangée par rapport à la notation orientée-objet, les attributs d'un agent étant l'union des attributs qui lui sont propres avec les attributs de la superclasse.

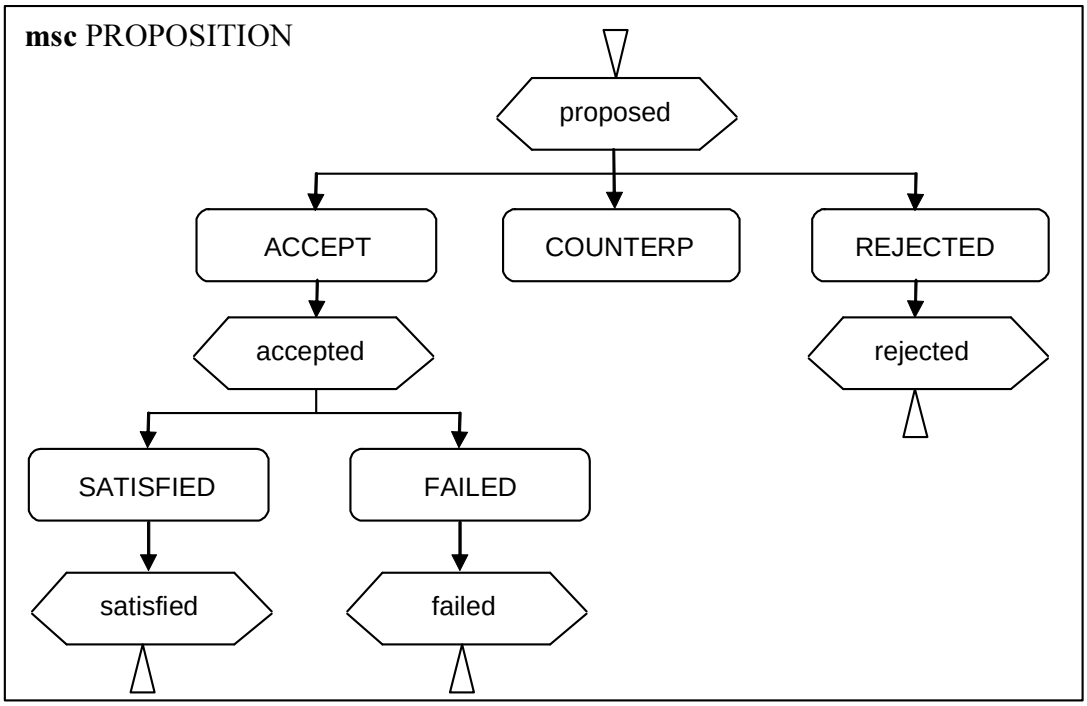


FIG. 3.9 – Protocole de coopération en HMSC.

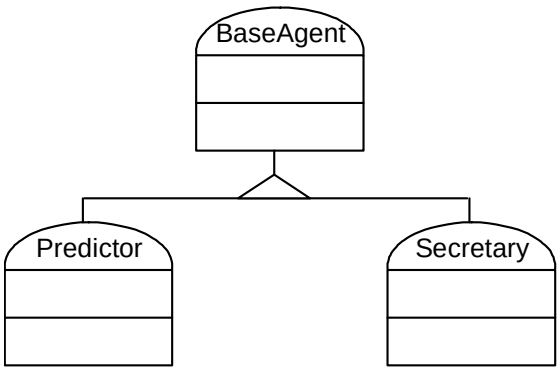


FIG. 3.10 – Exemple de modèle d'organisation.

Le modèle d'expertise

Le modèle d'expertise de MAS-CommonKADS est identique à celui de CommonKADS, il s'agit donc de modéliser l'ensemble des connaissances du domaine d'application, indépendamment de la découpe en agents de celui-ci. Toutefois, comme nous l'avons indiqué dans la description du *modèle d'agents*, certains agents peuvent être spécialisés dans certaines tâches et connaissances et donc se référer à certaines instances du modèle d'expertise.

L'élaboration du modèle d'expertise consiste à modéliser trois types de connaissances :

- les inférences sur le domaine (par exemple, comment prédire le délai d'un vol)
- les raisonnements d'un agent (le modèle d'agent fait référence à ces instances du modèle)
- les inférences sur l'environnement (par exemple, comment un agent peut interpréter un événement provenant d'un autre agent ou du monde extérieur)

Ceci débouche sur la constitution de plusieurs instances du modèle, dont certaines peuvent servir à définir les autres, grâce à la grande réutilisabilité du modèle. La figure 3.3 présente un exemple d'instance du modèle d'expertise.

Le modèle de tâches

Les tâches sont décomposées grâce à un arbre "et/ou", les noeuds "ou" correspondant à une tâche dont les fils sont les différentes méthodes permettant d'effectuer cette tâche, les noeuds "et" correspondant à une méthode dont les fils sont les différentes sous tâches nécessaires à l'exécution de la méthode. Le modèle de tâches permet donc d'obtenir une vue statique et compositionnelle des tâches, alors que le modèle d'expertise fournit une vue dynamique des tâches en spécifiant l'enchaînement des sous tâches.

Le modèle de communications

Le modèle de communications reste également inchangé par rapport à CommonKADS. Celui-ci permet de détailler les interactions homme-machine, et les facteurs humains dont il faut tenir compte pour développer les interfaces utilisateur.

3.2.3 Design

Le but de la phase de design est de développer le *modèle de design* à partir des différents modèles élaborés à la phase d'analyse. Ceci consiste à élaborer :

- le *design de l'application* : il est ici question de décomposer le système en sous-modules, ces sous-modules étant généralement des agents. Il peut être également question d'identifier de nouveaux agents, notamment grâce aux modèles d'expertise et de tâches. En effet, lorsqu'une tâche peut être résolue grâce à différentes méthodes (cfr. notion de *Task Structure*), on peut attribuer ces différentes méthodes à différents agent, et ajouter un "agent de support" qui choisira à l'exécution quel agent résoudra la tâche.
- le *design de l'architecture* : celui-ci consiste à déterminer l'architecture du système multi-agent. Cette architecture est composée de trois niveaux, comme proposé dans [27] :
  1. le *niveau réseau*, niveau dans lequel l'infrastructure de l'architecture multi-agent est définie. Cette infrastructure peut consister en des agents jouant le rôle de serveurs de noms, de gestionnaires de groupe, et également en des protocoles réseau (http,...), des protocoles de sécurité, ... ;
  2. le *niveau des connaissances*, niveau dans lequel on décide si par exemple un gestionnaire d'ontologies sera utilisé, si l'on utilisera des ontologies distribuées, ... ;
  3. le *niveau de coordination*, niveau dans lequel on s'attache à améliorer la réutilisabilité des protocoles et des connaissances relatives à la coopération.
- le *design relatif à la plateforme* : à partir des différents choix effectués précédemment, on décide dans cette étape le système d'exploitation ainsi que le matériel qui sera utilisé

3.3 Evaluation

La méthodologie MAS-CommonKADS présente l'avantage de se baser sur des techniques qui ont prouvé leur efficacité, en l'occurrence CommonKADS ainsi que certains modèles provenant de l'orienté-objet et de l'ingénierie des protocoles. De plus, CommonKADS peut être vu comme un standard européen en matière de modélisation des connaissances. Un autre avantage de cette méthodologie réside dans le fait qu'elle permet de cerner deux concepts fondamentaux de la notion d'agent et de système multi-agent, en l'occurrence le comportement " intelligent " et le comportement social, et ceci grâce aux modèles d'expertise et de coordination.

Néanmoins, cette méthodologie n'est pas sans inconvénients. A nos yeux, l'inconvénient majeur de cette méthodologie est l'absence de processus d'élaboration clair et détaillé. En effet, bien que la méthodologie soit basée sur un modèle de cycle de vie du développement logiciel (Software Development Life Cycle) guidant le développeur au travers des étapes de conceptualisation, analyse, design, codage, test et maintenance, il n'y a pas de méthodes claires pour élaborer chacun des modèles de l'étape d'analyse. De plus l'élaboration de ces modèles doit se faire en parallèle, ce qui introduit un autre facteur de complexité. Toutefois, il faut de noter que la méthodologie définit l'état d'avancement de chacun des modèles (empty, identified, described et validated) afin de faciliter la gestion du projet [28].

Un autre inconvénient réside dans le fait qu'une fois les différents modèles constitués, il faut les traduire dans un langage de programmation, et que malheureusement il n'en existe aucun qui permette de traduire ces modèles de manière relativement aisée. Il faut donc traduire ces modèles dans un langage "traditionnel", tel que Java ou C++.

On pourrait finalement dire que la méthodologie MAS-CommonKADS, de par la richesse de ses modèles, permettrait de concevoir des systèmes complexes et de grande envergure, mais n'est pas très approprié pour concevoir *aisément* des systèmes de taille moyenne. Pour de tels systèmes, on préférera par exemple la méthodologie AAII, dont les modèles sont aisément transposable en 3APL.

Pour conclure, il est important de préciser que d'autres méthodologies ont tenté d'étendre CommonKADS à l'ingénierie de systèmes multi-agent, notamment la méthodologie CoMoMAS [16]. Cette méthodologie présente un modèle d'expertise plus spécifique aux systèmes multi-agent, notamment en y incluant des "connaissances réactives". Toutefois, le modèle de coordination semble moins élaboré que celui de MAS-CommonKADS.

Conclusions

Les méthodologies qui ont été présentées dans les deux chapitres précédents sont inspirées des méthodologies orientées-objet et de l'ingénierie des connaissances. Toutefois, d'autres approches existent, notamment celles basées sur l'expérience acquise lors du développement de systèmes multi-agent par le passé. Cette expérience a permit de dégager des principes généraux pour le développement de tels systèmes [28].

Il est intéressant de se demander si l'avantage des systèmes multi-agent présenté dans l'introduction, en l'occurrence celui de gérer la complexité de l'ingénierie du logiciel due aux interactions complexes entre composants, est réellement mis en oeuvre par ces méthodologies. Pour ce qui est de la méthodologie AAII, bien qu'elle comprenne un modèle d'interaction, elle ne fournit aucune norme sur l'élaboration de celui-ci. Toutefois, on pourrait imaginer d'y intégrer le modèle de coordination de MAS-CommonKADS, ou encore utiliser le protocole FIPA. Nous discuterons de cette possibilité dans le quatrième partie. La méthodologie Gaia fournit quant à elle une structure au modèle d'interaction. Toutefois, ce modèle d'interaction nous paraît beaucoup moins élaboré que celui de MAS-CommonKADS, le formalisme utilisé ne permettant pas de décrire des protocoles de communication complexes, tels que la coopération et la négociation.

D'un point de vue plus pragmatique, il est intéressant de s'interroger sur la réelle applicabilité de ces méthodologies. La méthodologie AAII a été développée sur la base de l'expérience gagnée suite au développement d'un système de gestion de trafic aérien [37], dont certains éléments nous ont servi d'exemple lors de la description de cette méthodologie. Précisons que ce système de gestion de trafic aérien (OASIS) a été développé grâce au PRS (Procedural Reasoning System), sur lequel a également travaillé l'Australian Artificial Intelligence Institute. La méthodologie Gaia quant à elle a permis le développement d'un système multi-agent de gestion d'un business process de British Telecom, dont une brève étude de cas est développée dans [51]. Et enfin, la méthodologie MAS-CommonKADS a été utilisée dans plusieurs projets de recherche, notamment pour la création d'un système intelligent de

gestion de réseau [28]. Mais on remarquera que ces différents projets restent confinés au monde de la recherche, or le but des méthodologies qui ont permis leur élaboration est de permettre le développement de systèmes multi-agents dans l'industrie. A notre avis, la difficulté pour ces méthodologies de percer dans le monde de l'entreprise provient de la quasi absence de langage de programmation orientée-agent abouti, ainsi que de standard pour la définition de ceux-ci. Il est en effet très peu utile d'obtenir une spécification multi-agent d'un système au terme de l'application d'une des méthodologies présentées ci-dessus, s'il faut la traduire en des concepts orientés-objet pour l'implémenter en Java par exemple. Dans ce cas, le but des méthodologies de réduire l'effort d'implémentation n'est pas vraiment rempli. Toutefois, il est important de préciser que de nombreuses recherches sur l'élaboration de langages de programmation orientés-agent sont en cours, dont certains sont présentés dans la troisième partie. De plus, les méthodologies présentées ici ont permis de dégager un niveau conceptuel pour l'analyse de systèmes multi-agent, indépendant des langages de programmation utilisés, ce niveau conceptuel étant composé généralement de modèles d'agents et de modèles de groupes et de sociétés [28].

Malgré les différentes critiques adressées à ces différentes méthodologies, l'existence de celles-ci, comme le précisent [52] et [28], est une condition sine qua none pour l'introduction du développement de systèmes multi-agents dans l'industrie.

Deuxième partie

Méthodes formelles de conception orienté-agent

Chapitre 4

L'architecture BDI pour la spécification de systèmes multi-agents

4.1 Introduction

Dans ce chapitre, nous présenterons le formalisme BDI fourni par Rao et Georgeff [41], permettant de spécifier des agents en tant que *systèmes intentionnels*. Un agent intentionnel est un agent auquel on a attribué des états mentaux, tels que les croyances, les désirs, les intentions, ... Les théories rendant compte de l'interaction des différents états mentaux d'un agent sont appelées *théories agent* (*agent theories*), dont l'architecture BDI fait partie.

Dans la suite de cette introduction, nous allons examiner quelles peuvent être les différentes notions intentionnelles dont peut disposer un agent et la classification de celles-ci. Nous allons ensuite montrer pourquoi la logique traditionnelle (logique propositionnelle, logique des prédicats du 1er ordre, ...) est inapte à raisonner sur ces notions intentionnelles.

4.1.1 Notions intentionnelles

Comme nous l'avons dit précédemment, un agent peut être vu comme un système intentionnel. Il s'agit en fait d'une vue plutôt anthropomorphique du concept d'agent, qui consiste à lui attribuer des attitudes tels que les croyances, les connaissances, les intentions, ...

Plusieurs réflexions philosophiques ont été émises sur la réelle nécessité d'attribuer des attitudes humaines à un système informatique, et certaines d'entre-elles en sont arrivées à la conclusion que ces attitudes sont utiles lorsqu'elles sont attribuées à des entités dont la structure est complexe [38]. Ainsi, comme l'indique [50], il peut paraître absurde de considérer un système simple tel qu'un interrupteur comme un système possédant des croyances et des buts, étant donné que son fonctionnement peut être entièrement compris grâce à une description mécanique de celui-ci. Toutefois, pour des systèmes complexes tels qu'un agent ou un ordinateur, il est intéressant de raisonner sur ceux-ci en leur attribuant des notions intentionnelles, qui constituent dans ce cas-ci des outils d'abstraction [50].

Les attitudes intentionelles sont divisées, selon Shoham et Cousins [46] en trois catégories : les attitudes *informationnelles*, de *motivation* et *sociales*.

Les attitudes *informationnelles* sont généralement les croyances et les connaissances. Les attitudes de *motivation*, ou pro-attitudes, peuvent être les désirs, les intentions, les choix, ... Les attitudes *sociales* sont liées aux attitudes de motivation, mais elles impliquent d'autres agents. Ces attitudes sociales peuvent être les obligations, les permissions, ...

L'architecture BDI présentée dans ce chapitre considère qu'un agent est doté de trois attitudes : les Croyances, les Désirs (ou goals) et les Intentions. La présentation que nous ferons de cette architecture se base sur [41]. Il est important de préciser que l'architecture BDI considère l'intention comme une attitude primordiale d'un agent, contrairement à d'autres architectures dans lesquelles les intentions peuvent être exprimées en termes de croyances et de buts [6].

D'un point de vue sémantique, l'architecture BDI étend le modèle des mondes possibles, et d'un point de vue syntaxique, elle étend la logique modale temporelle. Nous allons présenter ces deux extensions dans la section 4.2 et 4.3. Mais avant toute chose, il est intéressant d'examiner pourquoi une logique traditionnelle n'est pas adéquate pour raisonner sur les attitudes intentionelles.

4.1.2 Inadéquation de la logique traditionnelle

Pour exprimer l'inadéquation de la logique traditionnelle à la formalisation des attitudes intentionnelles, nous allons reprendre l'exemple fourni dans [50] (repris de [14]). Supposons que l'on veuille exprimer l'affirmation suivante dans la logique du premier ordre :

"Janine croit que Cronos est le père de Zeus" (4.1)

Ceci pourrait donner :

$Bel(Janine, Father(Zeus, Cronos))$ (4.2)

Cette expression est invalide pour deux raisons [50]. La première est d'ordre syntaxique : l'expression (4.2) n'est pas une formule bien formée (wff) de la logique des prédicats du premier ordre, car Father(Zeus,Cronos) est un prédicat et le prédicat $Bel/2$ ne peut contenir que des termes. La deuxième raison est d'ordre sémantique : supposons que l'on ajoute le prédicat suivant :

$(Zeus = Jupiter)$ (4.3)

Ceci peut être considéré comme vrai, étant donné que Zeus et Jupiter sont vus comme étant la même divinité. Ainsi, les règles de la logique du premier ordre permettant de dériver l'expression suivante :

$Bel(Janine, Father(Jupiter, Cronos))$ (4.4)

L'intuition considère que l'expression 4.4 est fausse.

Le problème général qui empêche la logique du premier ordre de représenter les notions de désir, croyance, ... vient du fait que ces notions sont "référentiellement opaques", c'est-à-dire que les règles de substitution de la logique classique ne s'y appliquent pas. En effet, dans la logique propositionnelle par exemple, la valeur de (p $\bigwedge$ q) dépend uniquement des valeurs de p et q. Par contre, lorsqu'on travaille avec des notions intentionnelles, la valeur de vérité de

"Janine croit que p" (4.5)

par exemple ne dépend pas uniquement de la valeur de vérité de p. En effet, Janine peut ne pas croire en quelque chose qui soit vrai.

Etant donnée l'inadéquation sémantique et syntaxique de la logique classique, une logique qui puisse rendre compte des notions intentionnelles doit se baser sur un modèle sémantique et un langage de formulation adéquats. Ainsi, l'architecture BDI se base sur la sémantique des mondes possibles (en y ajoutant une notion de "temps"), et sur le langage de formulation de la logique modale temporelle, présentés dans les deux sections suivantes.

4.2 La sémantique des mondes possibles

La sémantique des mondes possibles a été proposée pour la première fois par Hintikka [21]. Celui-ci considérait que les croyances d'un agent peuvent être vues comme un ensemble de mondes possibles. Afin d'expliquer ceci, nous allons reprendre l'exemple de [50]. Supposons qu'un agent joue au poker contre d'autres agents. Le but pour celui-ci est donc de connaître au maximum ce que contient le jeu de ses opposants. Une manière d'accomplir ce but est de calculer toutes les façons avec lesquelles le jeu initial a été distribué à chacun des joueurs. Supposons que chacune de ces façons soit inscrite sur un bout de papier. Il s'agit ensuite d'éliminer systématiquement chaque bout de papier qui est considéré comme impossible, étant données les connaissances de l'agent. Par exemple, si l'agent possède l'as de pique, celui-ci éliminera toutes les configurations dans lesquelles il ne possède pas l'as de pique. Ainsi, chaque bout de papier restant après ce processus est appelle "monde possible". Un autre terme souvent utilisé est le terme "alternatives épistémiques", qui désigne l'ensemble des mondes possibles étant données les croyances d"un agent. En d'autres termes, *une chose vraie dans toutes les alternatives épistémiques d'un agent peut être considéré comme une croyance de l'agent.*

La sémantique des mondes possibles est reprise par l'architecture BDI, mais celle-ci lui apporte certaines extensions, comme nous le verrons maintenant.

4.2.1 Extensions apportées par l'architecture BDI

L'architecture BDI apporte principalement deux extensions aux modèle des mondes possibles [41]. Premièrement, le modèle BDI décrit comme un ensemble de mondes possibles non seulement les croyances, main également les goals (ou désirs) et les intentions. Deuxièmement, le modèle BDI introduit une notion de temps. Ainsi, un monde est représenté comme une structure temporelle, appelée *arbre du temps* (*time tree*), alors que traditionnellement, un monde est représenté par un ensemble de croyances. Un exemple d'arbre du temps est présenté à la figure 4.1.

Un point particulier d'un arbre du temps dans un monde est appelé *situation*. Les arcs d'un time tree représentent des *événements*, directement exécutables par un agent. Ainsi chaque branche dans un arbre du temps représente un choix disponible pour l'agent à un moment donné.

Nous allons maintenant décrire le formalisme utilisé par le modèle BDI afin de rendre compte de ces structures.

4.3 Formalisme utilisé

Le formalisme utilisé par le modèle BDI est similaire au formalisme CTL (Computation Tree Logic) [11]. Une distinction sémantique est faite entre les *formules d'état* et les *formules de chemin*. Les premières sont évaluées à un point spécifié d'un arbre du temps, les deuxièmes le sont sur un chemin spécifié d'un arbre du temps.

Le formalisme CTL se base sur un langage modal temporel, dont les opérateurs sont $\bigcirc$ (suivant), $\Diamond$ (finalement),$\Box$ (toujours) et $\bigcup$ (jusqu'à ce que). D'autres opérateurs spécifiques aux formules de chemin sont utilisés. Ces opérateurs sont *optional*() (noté également $E()$) et *inevitable*() (noté également $A()$). Une formule de chemin est dite *optional*, si, à un point particulier du time tree, cette formule est vraie pour au moins un chemin partant de ce point. Une formule de chemin est dite *inevitable*, si, à un point particulier du time tree, cette formule est vraie pour tous les chemins partant de ce point. Par exemple, dans la figure 4.1, on peut dire *optional*($\Box r$), car la proposition r ("Mary vit en Australie") n'est vraie que sur une branche partant de la racine (*optional*), et elle est vraie à chaque instant dans la branche ($\Box$). On peut également dire *inevitable*($\Diamond q$), car la proposition q ("la fin du monde a lieu") est vraie en un point sur toutes les branches de l'arbre du temps (*inevitable*), et elle devient vraie à point donné de l'arbre du temps ($\Diamond$).

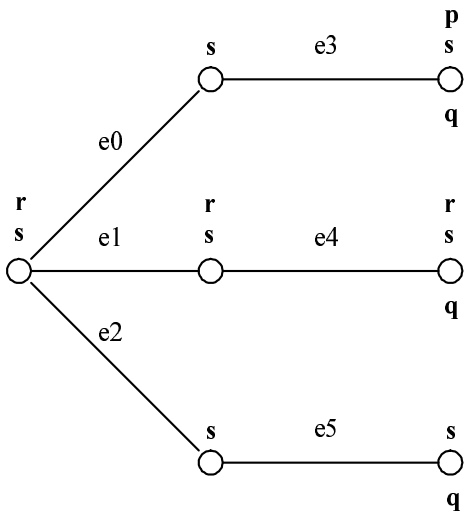


FIG. 4.1 – Exemple de time tree.

Le formalisme BDI étend ce langage modal temporel en y intégrant des formules de la logique du premier ordre, et également en y introduisant des opérateurs modaux pour les croyances, désirs et intentions. Nous allons maintenant décrire la syntaxe de ce formalisme de manière formelle.

Supposons un ensemble V de variables, et un ensemble E disjoint d'événements primitifs. Une formule d'état est définie de la manière suivante :

- Une formule du premier ordre est une formule d'état. Une formule du premier ordre a la forme $p(x_1, ..., x_n)$, où p est un symbole de prédicat et x_1, ..., x_n sont des variables ;
- si Φ1 et Φ2 sont des formules d'état, et x est une variable ou un événement, alors ¬Φ1, Φ1 $\bigvee$ Φ2, et ∃x Φ1(x) sont des formules d'état ;
- Si e est un événement, alors *succeeds(e), fails(e), does(e), succeeded(e), failed(e), et done(e)* sont des formules d'état ;
- Si Φ est une formule d'état, alors BEL(Φ), GOAL(Φ) et INTEND(Φ) sont des formules d'état ;
- Si Φ est une formule de chemin, alors optional(Φ) est une formule d'état

Une formule de chemin est définie de la manière suivante :

- Toute formule d'état est également une formule de chemin
- si $\Phi 1$ et $\Phi 2$ sont des formules de chemin, alors $\neg\Phi 1$, $\Phi 1 \bigvee \Phi 2$, $\Phi 1 \bigcup \Phi 2$, $\Diamond\Phi 1$, $\bigcirc\Phi 1$, sont des formules de chemin

Remarquons premièrement que les opérateurs *inevitable*() et $\Box$ ne figurent pas dans cette description. En fait ces opérateurs peuvent être définis à partir des autres opérateurs. En l'occurence, $inevitable(\psi) = \neg optional(\neg\psi)$, et $\Box\psi = \neg\Diamond\neg\psi$. Ensuite, précisons que les formules *succeeded(e)* et *failed(e)* représentent la performance qui vient d'être effectuée (respectivement réussie ou ratée), la formule *done(e)* représente la performance qui vient d'être effectuée, indépendamment de sa réussite ou de son échec. Les formules *succeeds(e), fails(e), does(e)* sont similaire, sauf qu'elles se réfèrent aux performances qui seront effectuées dans tous les futurs immédiats.

La section suivante présente la sémantique des différentes structures définies ci-dessus.

4.4 Sémantique

Dans cette section, nous fournirons tout d'abord le cadre général de la sémantique en donnant la définition d'une *interprétation*, d'un *monde* et d'un *sous-monde*, afin de pouvoir donner ensuite la sémantique des formules du premier ordre, des événements et des croyances, goals et intentions.

4.4.1 Cadre général

Définition 1 : Une interprétation M est un tuple, $M = (W, E, T, \prec, U, \beta, G, I, \Phi)$. W est un ensemble de mondes, E est un ensemble d'événements primitifs, T est un ensemble de *points du temps* (*time points*), $\prec$ est une relation d'ordre strict sur ces points (la relation $\prec$ organise donc les points en arbres), U est l'univers de discours, et Φ est une fonction qui assigne des symboles de prédicats à des éléments de U pour chaque monde et *point du temps*.
Une *situation* (un monde à un moment donné), est une paire w_t, où w est un monde et t est un *point du temps*. Les relations β, G et $I \subseteq W \times T \times W$ assignent à la situation courante de l'agent respectivement ses mondes de croyances, de goals et d'intentions accessibles. La relation R se réfère à n'importe laquelle des relation β, G, I. Ainsi, dans la figure 4.2, ${\beta_{t1}}^{w0} = \{b1, b2\}$.

Définition 2 : Chaque monde w de W, appelé *arbre du temps*, est un tuple $< T_w, A_w, S_w, F_w >$, où $T_w \subseteq T$ est l'ensemble des points du temps de w et où A_w est identique à $\prec$ restreint un points de T_w. Les fonctions S_w et F_w représentent intuitivement les arcs d'un *time tree*. Plus formellement, $S_w : T_w \times T_w \rightarrow E$ et $F_w : T_w \times T_w \rightarrow E$. Notons que les domaines de S_w et F_w doivent être disjoints, et que si $S_w(t_i, t_j) = S_w(t_i, t_k)$, alors $t_j = t_k$, et de même pour F. Ainsi, la fonction S représente l'événement qui s'est produit avec succès entre deux points, alors que la fonction F représente l'événement qui a abouti à un échec entre deux points.

Définition 3 : Un *sous-monde* est un sous-arbre d'un monde dont les formules ont la même valeur de vérité. Ainsi, w' est un sous-monde de w, noté $w' \sqsubseteq w$, si et seulement si :

1. $T_{w'} \sqsubseteq T_w$
2. $\forall$ u ϵ $T_{w'}$, $\Phi(q, w', u) = \Phi(q, w, u)$, où q est un symbole de prédicat
3. $\forall$ u ϵ $T_{w'}$, $R_u^w = R_u^{w'}$
4. $A_{w'}$ est A_w restreint aux *points du temps* de $T_{w'}$, et de même pour $S_{w'}$ et $F_{w'}$

4.4.2 Sémantique des formules du 1er ordre

Supposons une interprétation M, et une assignation de variable v dans l'univers de discours. v_d^i est la fonction qui assigne d à la variable i et qui est la fonction identique à v partout ailleurs. La sémantique des formules d'état peut être exprimé comme suit :

$M, v, w_t \models q(y_1, ..., y_n)$ ssi $(v(y_1), ..., v(y_n)) \epsilon \Phi[q, w, t]$ où q est un symbole de prédicat
$M, v, w_t \models \neg\phi$ ssi $M, v, w_t \not\models \phi$
$M, v, w_t \models \phi_1 \bigvee \phi_2$ ssi $M, v, w_t \models \phi_1$ ou $M, v, w_t \models \phi_2$
$M, v, w_t \models \exists i\phi$ ssi $M, v_d^i, w_t \models \phi$ pour un certain d dans U

La sémantique des formules de chemin est la suivante :

$M, v, (w_{t0}, w_{t1}, ...) \models \phi$ ssi $M, v, w_t \models \phi$ où ϕ est une formule d'état
$M, v, (w_{t0}, w_{t1}, ...) \models \bigcirc\psi$ ssi $M, v, (w_{t1}, ...) \models \psi$
$M, v, (w_{t0}, w_{t1}, ...) \models \Diamond\psi$ ssi $\exists k$, $k \geq 0$ tel que $M, v, (w_{tk}, ...) \models \psi$
$M, v, (w_{t0}, w_{t1}, ...) \models \psi_1 \bigcup \psi_2$ ssi :

- $\exists k$, $k \geq 0$ tel que $M, v, (w_{tk}, ...) \models \psi_2$ et $\forall j$, $0 \leq j < k$, $M, v, (w_{tj}, ...) \models \psi_1$, ou
- $\forall j \geq 0$, $M, v, (w_{tj}, ...) \models \psi_1$

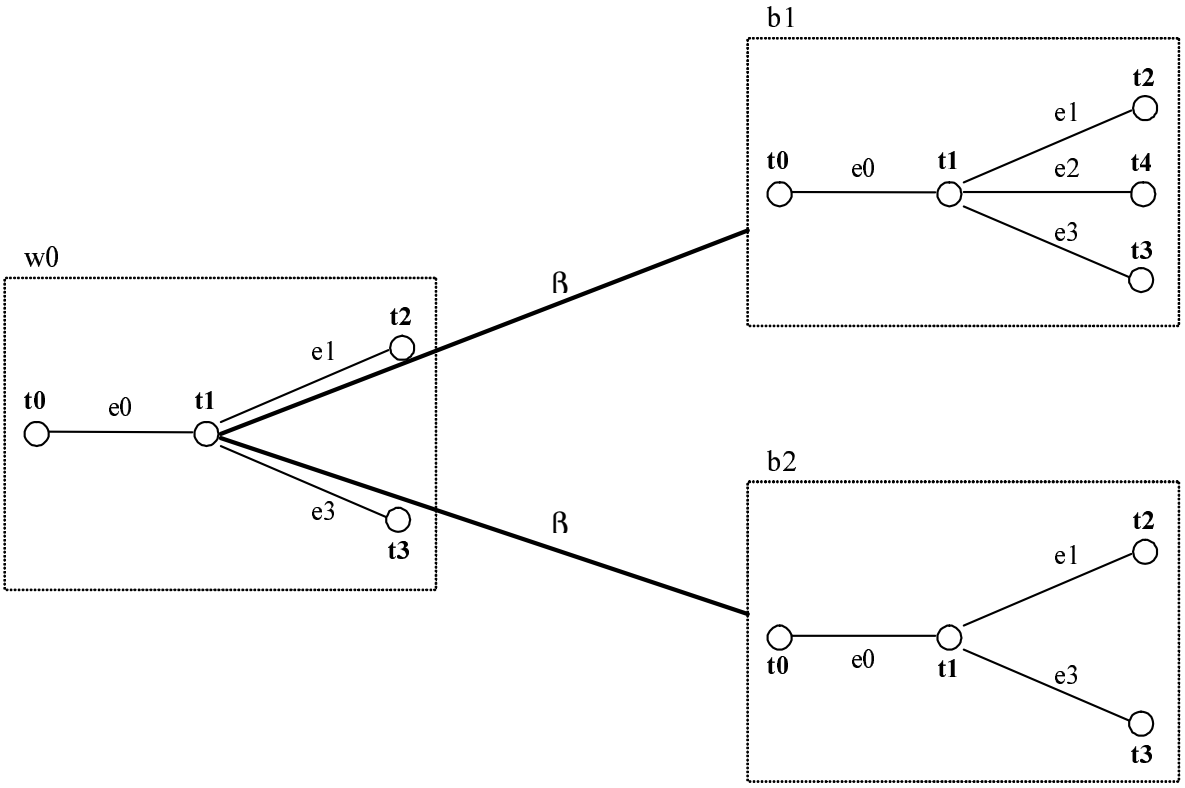


FIG. 4.2 – Relation d'accessibilité entre mondes.

Finalement nous pouvons définir la sémantique de l'opérateur *optional*() :

$M, v, w_{t0} \models optional(\psi)$ ssi il existe un chemin $(w_{t0}, w_{t1}, ...)$ tel que $M, v, (w_{t0}, w_{t1}, ...) \models \psi$

Une formule bien formée qui ne contient pas d'occurrence positive de l'opérateur *inevitable*() (ou d'occurrence négative de *optional*()) en dehors des parenthèses des opérateurs BEL, $GOAL$, $INTEND$, est appelée *O-formule*. A l'inverse, une *I-formule* est une formule qui ne contient pas d'occurrence positive de l'opérateur *optional*().

4.4.3 Sémantique des événements

Supposons une interprétation M, et une assignation de variables v dans l'univers de discours. La sémantique des formules exprimées sur des événements peut être exprimée comme suit :

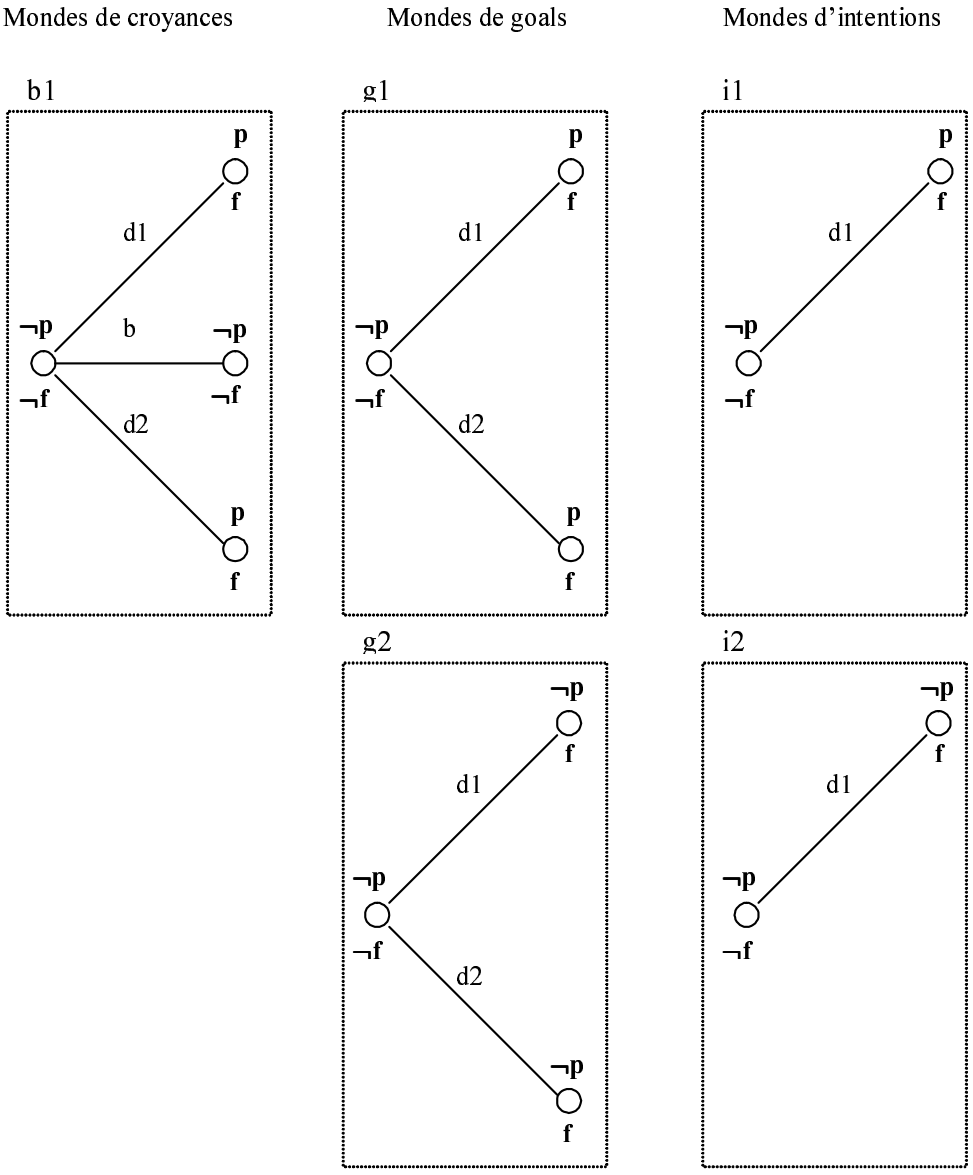
$M, v, w_{t1} \models succeeded(e)$ ssi $\exists t_0 \, \epsilon \, T_w$ tel que $S_w(t_0, t_1) = e$
$M, v, w_{t1} \models failed(e)$ ssi $\exists t_0 \, \epsilon \, T_w$ tel que $F_w(t_0, t_1) = e$
$done(e)$ est défini comme $succeeded(e) \bigvee failed(e)$;
$succeeds(e)$ est défini comme $inevitable(\bigcirc(succeeded(e)))$;
$fails(e)$ est défini comme $inevitable(\bigcirc(failed(e)))$
$does(e)$ est défini comme $inevitable(\bigcirc(done(e)))$

4.4.4 Sémantique des croyances, goals et intentions

Comme nous l'avons dit plus haut, l'architecture BDI se base sur le modèle des mondes possibles. Ainsi, à chaque situation (ou noeud dans le time tree du monde courant) est associée un ensemble de mondes de croyances accessibles, de mondes de goals accessibles et de mondes d'intentions accessibles. Le monde courant est généralement considéré comme étant l'un des mondes de croyances accessibles. Intuitivement, chaque monde de croyances accessible (grâce à la relation β) représente une "hypothèse" sur l'état réel du monde, c'est-à-dire sur les actions disponibles et leurs conséquences. Les mondes de goals accessibles (grâce à la relation G) peuvent être vus comme les sous-ensembles des mondes de croyances dans lesquels l'agent "désire" être. Les mondes d'intentions accessibles (grâce à la relation I) peuvent être vus comme les sous-ensembles des mondes de goals que l'agent décide de suivre (ou d'exécuter).

Ainsi, on peut voir qu'il existe une relation entre mondes de croyances, de goals et d'intentions. En effet, à chaque monde de croyance accessible doit correspondre un monde de goals accessibles qui est un sous-monde de ce monde de croyance. De même, à chaque monde de goals accessible doit correspondre un monde d'intention qui est un sous-monde de ce monde de goals. La notion de sous-monde a été définie à la (Def 3). Ceci empêche par exemple un agent d'adopter des goals irréalisables d'après ses croyances. Les auteurs [41] qualifient cette propriété grâce au terme *réalisme fort*. Formulé autrement, le *réalisme fort* impose que si un agent a le goal que p soit vraie, il croira p comme étant une option.

La figure 4.3 illustre ce fait : le monde d'intentions i1 est un sous-ensemble du monde de goals g1, qui lui-même est un sous-ensemble du monde de croyances b1. La condition précitée est donc respectée.



Evénements : d1 : aller chez le dentiste 1
d2 : aller chez le dentiste 2
b : faire des courses
Faits : p : douleur
f : dent plombée

FIG. 4.3 – Mondes de croyances, de goals et d'intentions accessibles.

Mais ceci n'empêche d'avoir d'autres mondes de goals ou d'intentions accessibles qui ne soient pas des sous-mondes du monde de croyances. La condition ne requiert que "un" sous-monde pour chaque monde de croyances et de goals. Ceci permet donc à un agent de croire en un fait comme étant inévitable sans toutefois être forcé de l'adopter comme un goal, et également de ne pas être forcé à mettre en oeuvre un goal grâce à une intention. La figure 4.3 illustre cette particularité : bien que l'agent croie (dans b1) que le fait d'avoir une dent plombée s'accompagne forcément d'une douleur, il peut néanmoins avoir le goal (si on se restreint au monde g2) d'avoir une dent plombée sans avoir le goal d'avoir mal.

Nous pouvons maintenant définir la sémantique des opérateurs modaux $BEL()$, $GOAL()$ et $INTEND()$.

Nous pouvons dire qu'un agent croit la proposition p à un instant t , noté $BEL(p)$, si et seulement si p est vrai dans tous les mondes de croyances accessibles de l'agent au temps t. Ceci est défini formellement de la manière suivante :

$$M, v, w_t \models BEL(p) \text{ ssi } \forall w' \in {\beta_t}^w : M, v, w'_t \models p$$

D'une manière similaire, un agent a le goal p si et seulement si p est vrai dans tous les mondes de goals accessibles de l'agent au temps t.

$$M, v, w_t \models GOAL(p) \text{ ssi } \forall w' \in {G_t}^w : M, v, w'_t \models p$$

Finalement, un agent aura l'intention p si et seulement si elle est vraie dans tous les mondes d'intention accessibles de cet agent.

$$M, v, w_t \models INTEND(p) \text{ ssi } \forall w' \in {I_t}^w : M, v, w'_t \models p$$

Notons que les intentions peuvent aussi bien porter sur des croyances, des goals, des intentions que sur des actions. Toutefois, l'usage naturel considère que les intentions portent seulement sur des actions.

Pour illustrer ces concepts, reprenons l'exemple de la figure 4.3. Nous pouvons voir premièrement que la formule $GOAL(inevitable(\Diamond f))$ est vraie au temps $t0$ – car la formule f est vraie à moment donné ($\Diamond$) sur toutes les branches ($inevitable$) partant du noeud $t0$, dans tous les mondes de goals accessibles – alors que l'on ne peut affirmer $GOAL(inevitable(\Diamond p))$. Ceci illustre bien ce qui a été dit plus haut, en l'occurrence que bien que l'agent croie que le plombage d'une dent soit toujours accompagné d'une

douleur, celui-ci n'est pas forcé d'avoir le but d'avoir mal. Ici se révèle donc l'intérêt d'avoir ajouté un monde de goal possible, qui empêche la formule $GOAL(inevitable(\Diamond p))$ d'être vraie.

Toujours dans l'exemple de la figure 4.3, la formule $INTEND(succeeds(d1))$ (ou $INTEND(inevitable(\bigcirc(succeeded(d1))))$ est vraie au temps $t0$, car $succeeds(d1)$ est vraie dans tous les mondes d'intentions accessibles au temps $t0$. On peut donc dire que l'agent a l'intention d'accomplir l'action $d1$ (littéralement, il a l'intention de faire réussir l'action $d1$). De plus, la notion de *réalisme fort* impose que $GOAL(optional(\bigcirc(succeeded(d1))))$ et $BEL(optional(\bigcirc(succeeded(d1))))$ soient vrais.

4.5 Axiomes

Dans cette section, nous fournirons les axiomes et les conditions sémantiques décrivant les relations entres Croyances, Goals et Intentions.

Compatibilité Croyance-Goal

La notion de *réalisme fort* entre goals et croyances expliquée plus haut peut être exprimée grâce à la condition suivante :

(CI1) $\forall w' \in \beta_t^w$, $\exists w'' \in G_t^w$ tel que $w'' \subseteq w'$

On peut alors exprimer cette condition par un axiome adéquat et complet :

(AI1) $GOAL(\alpha) \supset BEL(\alpha)$, α est une O-formule

Ceci exprime que si un agent adopte une O-formule comme goal, celui-ci doit croire en cette formule. Ainsi, par exemple, si dans tous les mondes de goals accessibles, il y a au moins un chemin sur lequel la proposition p devient finalement vraie, alors il doit également exister dans tous les mondes de croyances accessibles au moins un chemin sur lequel la proposition p devient finalement vraie.

Compatibilité Goal-Intention

La notion de *réalisme fort* entre intentions et goals peut être exprimée grâce à la condition sémantique CI2, de laquelle on a dérivé l'axiome AI2.

(CI2) $\forall w' \epsilon G_t^w$, $\exists w'' \epsilon I_t^w$ tel que $w'' \subseteq w'$
(AI2) $INTEND(\alpha) \supset GOAL(\alpha)$

D'un manière similaire à la compatibilité Croyance-Goal, la compatibilité goal-intention établit donc que si un agent adopte une O-formule comme intention, celui-ci doit avoir également adopté cette formule comme goal.

Relation Intentions-Actions

L'axiome décrit ici établit que si un agent a une intention sur une action primitive e, il effectuera cette action.

(AI3) $INTEND(does(e)) \supset does(e)$

Intuitivement, la condition sémantique sur laquelle est basé cet axiome établit que s'il n'existe qu'un seul choix d'action à effectuer à un moment donné dans tous les mondes d'intentions, et que ce choix soit le même dans tous ces mondes, l'agent sera forcé de l'accomplir.

Croyances à propos d'intentions

Si un agent a une intention, celui-ci croit qu'il a cette intention. Ceci est exprimé grâce à la condition et l'axiome suivants.

(CI4) $\forall w' \epsilon \beta_t^w$ et $\forall w'' \epsilon I_t^w$ on a $w'' \epsilon \beta_t^{w'}$
(AI4) $INTEND(\phi) \supset BEL(INTEND(\phi))$

Intuitivement, si la formule $INTEND(\phi)$ est vraie, cela veut dire que ϕ est vraie dans tous les mondes d'intentions. Comme tous les mondes d'intentions sont accessibles depuis chaque monde de croyances (CI4), la formule $INTEND(\phi)$ est vraie dans tous les mondes de croyances. La formule $BEL(INTEND(\phi))$ est donc vraie. La même justification intuitive s'applique aux deux axiomes suivants.

Croyances à propos de goals

La condition et l'axiome suivants établissent que si un agent a un goal ϕ, l'agent croit qu'il a ce goal a accomplir.

(CI5) $\forall w' \epsilon \beta_t^w$ et $\forall w'' \epsilon G_t^w$ on a $w'' \epsilon \beta_t^{w'}$
(AI5) $GOAL(\phi) \supset BEL(GOAL(\phi))$

Goals à propos d'intentions

La condition et l'axiome suivants établissent que si un agent a l'intention ϕ, celui-ci doit avoir le goal d'accomplir ϕ.

(CI6) $\forall w' \epsilon\ G_t^w$ et $\forall w'' \epsilon\ I_t^w$ on a $w'' \epsilon\ G_t^{w'}$
(AI6) $INTEND(\phi) \supset GOAL(INTEND(\phi))$

Conscience des événements primitifs

L'axiome suivant établit qu'un agent a conscience de tous les événements primitifs qui sont faits.

(AI7) $done(e) \supset BEL(done(e))$

Pas de report infini des intentions

L'axiome suivant établit qu'un agent ne peut reporter indéfiniment une de ses intentions. Ainsi, à un moment donné, celui-ci doit abandonner une de ses intentions.

(AI8) $INTEND(\phi) \supset inevitable(\Diamond(\neg INTEND(\phi)))$

Il est important de préciser que d'autres axiomes sont spécifiés dans [41], établissant les relations entre intentions actuelles et intentions futures. Nous ne détaillerons pas ces axiomes ici, étant donné qu'il sont spécifiques au type d'agent que l'on veut formaliser (*blindly committed, single minded, open minded*). De plus, les axiomes standards KD45 de la logique modale sont vrais pour les croyances. Pour ce qui est des goals et des intentions, les axiomes K et D sont vrais, c'est-à-dire que les goals et intentions doivent être fermés pour l'implication $(GOAL(\alpha \rightarrow \beta) \rightarrow (GOAL(\alpha) \rightarrow \beta))$, et consistants $(\neg(INTEND(\alpha) \wedge INTEND(\neg\alpha)))$. La règle d'inférence modale (ou règle de nécessité RN) est également vraie, c'est-à-dire qu'un agent croit toutes les formules valides, les a comme buts et également comme intentions. Ceci provoque le problème bien connu de l'*omniscience logique*. En effet, un agent ne peut en pratique connaître toutes les formules valides qui sont des conséquences de celles qu'il connaît, car cet agent a des ressources limitées (temps de calcul, mémoire, ...).

4.6 Conclusions

La première question que l'on peut se poser est de savoir si ce formalisme permet la spécification de système réels. La sémantique sous-jacente à ce formalisme étant celle des mondes possibles, cela revient à se demander si celle-ci s'applique à des cas concrets.

Reprenons l'exemple du système de gestion de trafic aérien OASIS [39], qui a été utilisé pour décrire la méthodologie AAII dans le chapitre 2. L'agent *aircraft* a la responsabilité de voler le long d'un chemin en suivant certains *waypoints*. Toutefois, il existe une certaine incertitude provenant des données sur le vent. Une manière de spécifier cet agent grâce au formalisme présenté ici serait de modéliser un monde de croyance accessible pour chaque valeur de la vitesse du vent à un waypoint donné. Les différentes branches possibles à un moment donné seraient donc les choix de trajectoires, d'altitude et de vitesse pour l'agent étant donnée les données sur le vent. Les mondes de goals accessibles seront les mondes de croyances desquels on a éliminé les mondes pour lesquels le dernier waypoint n'est pas l'aéroport. En effet, le but ultime de l'agent *aircraft* est d'atterrir à l'aéroport. Les mondes d'intentions accessibles sont déduits des mondes de goals accessibles en ne retenant que les chemins optimaux. Ainsi, on peut voir que la sémantique des mondes possibles permet de formaliser avec élégance certains problèmes. Notons que la spécification en elle-même ne fait pas référence à ces mondes possibles, elle ne se fait généralement qu'au moyen de formules présentées au point 3.3. Remarquons également que ce type de spécification respecte la condition de ne donner aucun détail sur l'implementation du système. En effet, elle ne fournit que le comportement attendu de ce système.

Néanmoins, la spécification sert généralement de support à l'implémentation. Ainsi, selon [52], trois méthodes sont possibles lorsqu'il s'agit de transformer une spécification abstraite en un modèle de calcul concret.

Premièrement, la technique la plus utilisée est le raffinement manuel de la spécification en une forme exécutable grâce à des méthodes informelles. Ceci consiste généralement à décomposer la spécification en un ensemble de spécifications moins abstraites qui satisfont la spécification originale. Ce processus est répété jusqu'à ce que les spécification soient assez simples pour être implémentées.

Une deuxième approche consiste à compiler la spécification abstraite grâce à une technique de traduction automatique. Ceci a l'avantage par rapport à la technique que nous présenterons ci-après d'avoir une plus grande efficacité au moment de l'exécution. En effet, une telle technique permet de

"déplacer" le raisonnement symbolique sous-jacent à la spécification hors de l'exécution. Cette technique dépend en fait de la relation étroite entre le modèle des mondes possibles et les machines à états finis [52]. La compilation consiste en fait à établir une *preuve* de l'implémentabilité de la spécification, en construisant un modèle qui satisfasse cette spécification. Si ce modèle existe, la spécification est satisfaisable, sinon elle ne l'est pas. A partir de ce modèle, il est possible d'obtenir un automate qui implémente la spécification. Nous ne discuterons pas ici des techniques de preuves et de compilation de spécifications (pour plus de références, voir [52]) .

Certains problèmes sont néanmoins inhérents à cette technique. En effet, la plupart des formalismes de spécification utilisent la logique du premier ordre, qui est non-décidable, ce qui rend impossible que la compilation réussisse sur tous les problèmes. De plus, ce processus de compilation aboutit généralement à une machine à états finis, moins puissante qu'une machine de Turing, et qui ne peuvent modifier leur comportement à l'exécution.

La dernière approche consiste à exécuter directement une spécifcation. Par rapport à la compilation de spécifications, il s'agit ici de disposer d'un *interpréteur* de spécifications. Cette technique consiste également à prouver la satisfaisabilité de la spécification en construisant (au moment de l'exécution cette fois) un modèle qui satisfasse la spécification. Toutefois, l'implémentation directe de spécifications exprimées grâce à une logique complexe (telle que celle du formalisme BDI) est difficilement réalisable. En effet, le problème de l'omniscience logique du formalisme BDI impose qu'un agent dispose de capacités de raisonnements infinies, ce qui est impossible à l'exécution. De plus, établir une preuve de la spécification au moment de l'exécution peut prendre beaucoup de temps, et diminuer ainsi la réactivité d'un agent. Notons toutefois que Rao et Georgeff proposent une technique alternative pour l'interprétation de spécifications BDI [39], dont le fonctionnement est similaire à l'interpréteur d'AgentSpeak(L) présenté dans le chapitre 6.

Dès lors, plusieurs langages de programmation orientés-agent qui permettent l'implémentation directe de spécification se basent sur un sous-ensemble de la logique BDI, tels que AgentSpeak(L) et 3APL, ou sur d'autres formalismes de spécification calculables. La partie suivante présentera ces différents langages de programmation. Néanmoins, le but de ce mémoire étant d'étudier les méthodes et outils permettant la conception de SMA, nous ne nous focaliserons que sur l'utilisation de ces langages, et non sur leur mécanisme d'exécution des spécifications.

Troisième partie

La programmation orientée-agent

Chapitre 5

Le langage de programmation 3APL

5.1 Introduction

Dans le but d'appliquer de manière effective la technologie BDI ainsi que les méthodes de conception orientées agent, un certain nombre de langages de programmation ont été proposés tel que : Agent-0 [45], PLACA [48], AgentSpeak(L) [40], 3APL [20]... Toutefois, des langages de programmation orientés agent basés sur d'autres théories existent aussi, tel que ConGolog [15].

Dans cette troisième partie, nous avons donc choisi de présenter trois langages de programmation orientés agent assez représentatifs. Notre choix s'est donc porté sur les langages 3APL (chapitre 5) , AgentSpeak(L) (chapitre 6) et ConGolog (chapitre 7). Les questions liées à la sémantique opérationnelle et à la théorie des preuves de ces différents langages ne seront pas abordées ici. Des informations complémentaires sur ces sujets sont disponibles dans [33], [18].

Dans ce chapitre, nous allons donc commencer par aborder le langage 3APL. 3APL est un langage de programmation orienté-agent créé par Koen Hindriks [20] dans le cadre de sa thèse à l'université d'Utrecht. Ce langage a été implémenté en C et C++ par Steven Anker. 3APL est en fait l'abréviation de "*An Abstract Agent Programming Language*". L'originalité de ce langage est qu'il combine de la programmation à la fois impérative et logique, puisqu'il nous est donné la possibilité d'utiliser aussi bien des instructions IF-THEN-ELSE et WHILE que des prédicats Prolog.

La motivation principale pour développer ce langage de programmation orienté-agent consiste à fournir une API (Application Programming Interface) qui permet d'écrire des programmes orientés-agent en terme d'agent et non pas en terme d'objet ou de procédure. Ce langage a pour but premier de faciliter la programmation des comportements complexes des agents.

Dans un premier temps, nous introduirons l'agent 3APL dans ses grandes lignes avant d'en présenter les concepts fondamentaux que sont : les croyances (beliefs), les actions de base (basic actions), les buts (goals) et les practical reasoning rules. Dans un deuxième temps, nous aborderons le fonctionnement plus concret du langage 3APL avec ses instructions impératives, son modèle d'exécution et sa grammaire. Ensuite, nous illustrerons les concepts vus par un exemple utilisant le langage 3APL. Nous terminerons avec la partie qui a fait l'objet de notre stage au département informatique de l'université d'Utrecht et qui concerne l'implémentation de la base des croyances (beliefBase) 3APL et de moyens simples de communication entre agents 3APL, à l'aide des langages de programmation XSB-PROLOG [23] et JAVA [22].

Différentes sources nous ont permis de réaliser ce chapitre dont principalement les articles suivant [20], [19], [9].

5.2 L'agent 3APL

3APL définit un agent comme une entité dotée de quatre concepts ou attributs fondamentaux : les "*croyances*", les "*buts*", les "*practical reasoning rules*" et les "*actions de base*" [19].

Chaque agent possède son propre interpréteur, utilisant un moteur prolog, qui détermine le comportement de l'agent.

Afin de correspondre à l'architecture présentée dans la figure 1.1, l'architecture d'un agent 3APL peut être schématisée de la manière présentée à la figure 5.1.

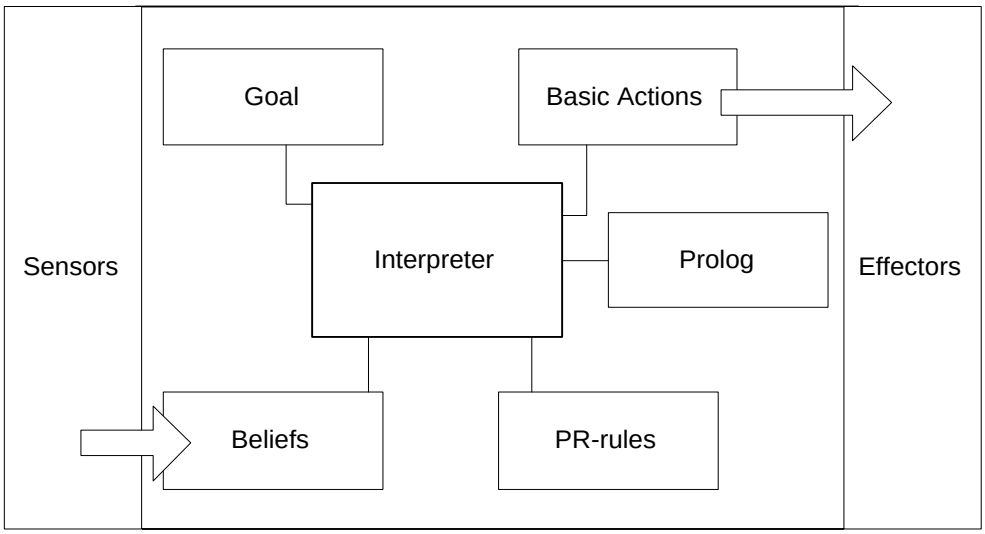


FIG. 5.1 – Architecture d'un agent 3APL

Un agent possède des ***croyances*** qui représentent les faits considérés comme vrais à un moment donné par l'agent. Ces croyances sont soit données au départ, soit acquises par l'agent grâce à ses senseurs. Cela implique que cet agent est capable d'interagir avec son environnement, c'est-à-dire qu'il a une interface d'entrée correspondant aux senseurs et une interface de sortie correspondant aux effecteurs.

Chaque agent agit sur son environnement à l'aide de ***buts*** qu'il doit tenter de satisfaire au mieux. Ces buts permettent la mise à jour continue de l'environnement, en ajoutant ou retirant des croyances.

Les ***practical reasoning rules*** (PR-rules) modifient les buts pour fournir aux agents les buts les plus appropriés à exécuter à un moment donné. Elles sont utilisées pour définir les capacités de réflexion des agents, pour déterminer un mécanisme efficace de révision des buts et pour établir des "plans". Elles sont donc capables de créer de nouveaux buts pour gérer des situations particulières. Ces règles de comportement transforment donc de simples agents en agents dits "intelligents" qui peuvent modifier eux-mêmes leurs propres croyances et buts. Ce principe ne s'applique pas aux PR-rules elles-mêmes qui restent statiques.

Les ***actions de base*** représentent les actions élémentaires que peut accomplir un agent, elles définissent donc l'expertise d'un agent. Ce sont ces actions de base qui permettent à l'agent d'agir sur son environnement grâce à ses effecteurs. Un agent peut donc être doté d'actions de base telles que "ramasser un objet", "déposer un objet" mais aussi "envoyer des messages

à un autre agent", ce qui, dans le cadre d'un groupe d'agents évoluant dans un environnement commun, permet une coordination entre ces agents pour que chacun réalise au mieux son(ses) but(s).

Tout comme les PR-rules, les actions de base ne peuvent pas être modifiées. Elles sont considérées comme des buts mais avec un statut spécial, car elles sont statiques contrairement aux buts proprement dits. Ces actions de base ne sont donc pas modifiables via les PR-rules.

Les croyances et les buts représentent clairement l' "***état mental***" d'un agent qui est mis à jour lors de son exécution.

Comme présenté dans la figure suivante, un agent est donc composé de trois couches : la première couche comprend les croyances, la seconde les buts ainsi que les actions de base, et la troisième les PR-rules.

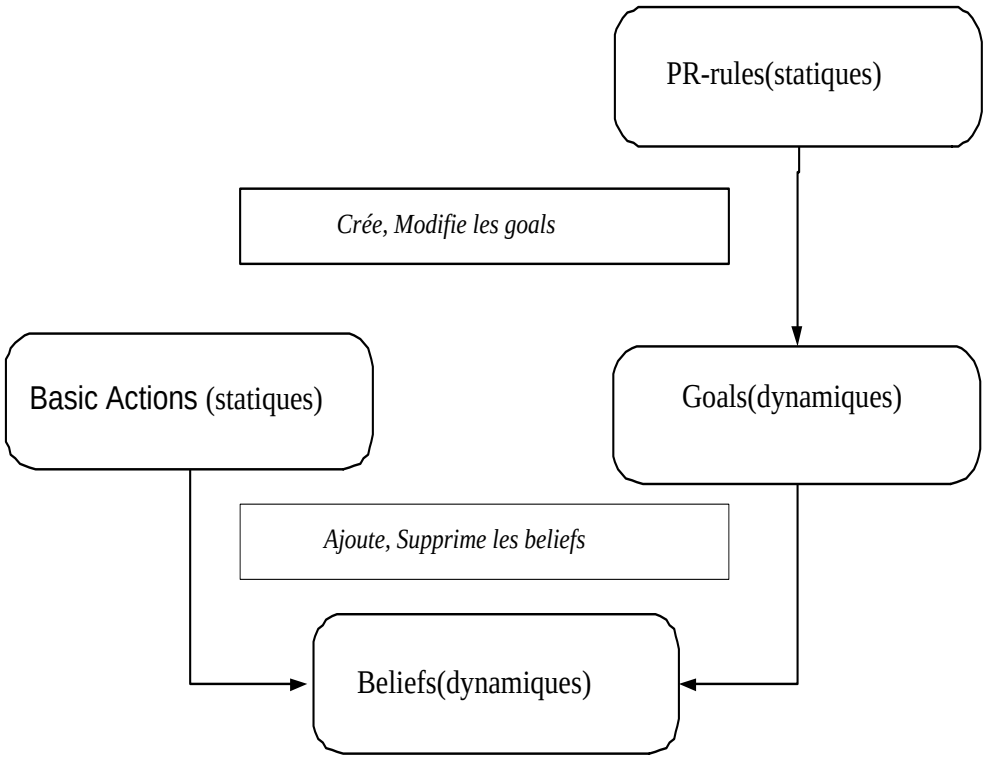


FIG. 5.2 – Couches d'un agent 3APL

On représente donc un agent par un quadruplet $< name, \sigma, \Pi, \Gamma >$
Où
- name = nom de l'agent
- σ = ensemble des croyances = "beliefBase"
- Π = ensemble des buts et ensemble des actions de base = "goalBase"
- Γ = ensemble des PR-rules = "ruleBase"

5.3 Les croyances 3APL

En principe, un croyance peut être représentée par n'importe quel langage de représentation de connaissance. Au niveau de 3APL, un croyance est représentée par une formule ne contenant aucune variable libre et appartenant à un langage L de la logique du premier ordre dont la définition est la suivante :

Si
- ***Pred*** = *l'ensemble de prédicats*
- ***Func*** = *l'ensemble de fonctions*
- ***Var*** = *l'ensemble de variables*
- ***Term*** = *l'ensemble de termes*

Alors le langage L se définit comme suit :
- Si ($p \in$ ***Pred***) et (arité(p)=n) et ($t1, ..., tn \in$ **Term**)
  alors $p(t1, ..., tn) \in L$
- Si $t1, t2 \in$ **Term**, alors $t1 = t2 \in L$
- Si $\Phi \in L$, alors $\neg\Phi \in L$
- Si $\Phi \in L$ et $\Psi \in L$, alors $\Phi \wedge \Psi \in L$
- Si $\Phi \in L$ et $\Psi \in L$, alors $\Phi \vee \Psi \in L$
- Si $\Phi \in L$ et $x \in Var$, alors $\forall x(\Phi) \in L$
- Si $\Phi \in L$ et $x \in Var$, alors $\exists x(\Phi) \in L$

L'ensemble des croyances d'un agent est appelé base des croyances (beliefBase). Cette base des croyances est mise à jour grâce à l'exécution des actions de base ou des buts; de plus, elle peut contenir des informations soit véridiques soit erronées sur l'environnement de l'agent. Les fonctions principales d'une base des croyances sont donc de stocker les croyances de l'agent et de gérer les requêtes ayant pour sujet les croyances qu'elle contient.

5.4 Les actions de base 3APL

5.4.1 Représentation des actions de base

Une action de base est représentée et déclarée de la manière suivante : $\{\gamma\}\alpha\{\theta\}$.

Où

- γ est un ensemble de préconditions sous la forme de formules de la logique des prédicats qui doivent être vraies pour que α soit applicable ;
- α est le nom de l'action ;
- θ est un ensemble de postconditions sous la forme de formules de la logique des prédicats qui sont vraies après l'exécution de l'action de base α et qui indiquent les changements qui seront apportés à la base des croyances après cette exécution.

Il faut souligner le fait que, dans cette représentation, les préconditions peuvent n'être ni vraies ni fausses si aucune indication n'est fournie sur celles-ci au niveau de la base des croyances.

5.4.2 Définition des actions de base

Si ***Asym*** *est l'ensemble des noms d'actions défini par les déclarations*
Alors ***Bact*** *est l'ensemble des actions de base défini par la règle suivante :*

Si a $\in$ **Asym** et t1 , . . . , tn $\in$ **Term** et n = arité de a
Alors a(t1,...,tn)$\in$ **Bact**

5.5 Les buts 3APL

Dans 3APL, les buts sont considérés selon une optique procédurale. En effet, l'objectif est de pouvoir, de la même manière qu'avec un langage impératif, être capable à partir de buts élémentaires d'établir des buts plus complexes.

L'ensemble des buts d'un agent est appelé la base des buts (goalBase). En principe, la base des buts de départ n'est constituée que d'un but (convention prise dans 3APL) placé à la base d'une structure de pile. Mais, afin de satisfaire ce but, il est décomposé, si possible, par les PR-rules en plusieurs buts de complexité inférieure. Ces PR-rules génèrent donc de nouveaux buts qui vont remplacer dans la pile l'ancien but. L'exécution d'un agent s'arrêtera lorsque cette pile de buts sera vide ou que le but au sommet de la pile

ne pourra plus être exécuté (action de base) ou décomposé. Il faut toutefois noter qu'un agent 3APL pourrait dès le départ posséder plusieurs buts, mais le manque de prioritisation entre les buts est une limite du langage 3APL pour simuler le comportement d'un agent.

5.5.1 Classification des buts :

Les buts (goals) sont classifiés au départ suivant trois *types de base* [20] :

1. Test Goals
   Un Test goal interroge la base des croyances mais permet aussi de calculer les valeurs ou de lier les variables libres du test. Ce but correspond donc à une opération d'initialisation de variables.

2. Achievement Goals
   Un achievement goal est utilisé pour décrire un état que l'agent voudrait atteindre. Ces achievement goals sont des formules atomiques du langage du premier ordre L.

   Atom =
   – ensemble des Achievement Goals
   – ensemble des formules atomiques du langage du premier ordre L

3. Actions de base
   Une action de base, comme cela a déjà été présenté, est assez particulière dans le sens qu'elle représente une capacité de l'agent et que, contrairement aux deux autres types de but, elle ne peut pas être modifiée via les PR-rules. Elle identifie donc les actions que peut directement réaliser l'agent. Une action de base ou capacité est un but qui, lorsqu'un agent l'exécute, peut changer l'état de l'environnement et modifier l'état mental de l'agent.

A partir de ces trois types de base, il est possible, en combinant ceux-ci, de construire des buts dits de *type complexe*. Ces combinaisons s'opèrent grâce aux opérateurs de séquence (;), de choix non-déterministe (+) et de parallélisme(||).

Si $\Pi 1, \Pi 2$ sont des buts
Alors $(\Pi 1; \Pi 2)$ et $(\Pi 1 + \Pi 2)$ et $(\Pi 1 || \Pi 2)$ sont des buts de type complexe

5.5.2 Définition des buts :

L'ensemble des buts (Goal) est donc défini inductivement par les 4 assertions suivantes [20] :
- Bact $\subseteq Goal$
- Atom $\subseteq Goal$
- Si $\alpha \in L$, alors $\alpha? \in Goal$
- Si $\Pi 1, \Pi 2 \in$ Goal , alors $(\Pi 1; \Pi 2)$ et $(\Pi 1 + \Pi 2)$ et $(\Pi 1 || \Pi 2) \in$ Goal

5.6 Les Practical Reasoning rules 3APL

L'ensemble des PR-rules d'un agent est appelé base des règles (ruleBase). Cette base des règles définit le comportement de l'agent. En outre, les PR-rules qui appartiennent à cette base des règles déterminent les prises de décisions et les choix de l'agent. On y retrouve spécifiée, pour chaque situation différente, la façon dont l'agent doit réagir. Les PR-rules constituent donc la majeure partie du travail de programmation des agents.

5.6.1 Représentation des PR-rules :

Une PR-rule est typiquement représentée par une expression de la forme suivante : $head \longleftarrow guard \mid body$

Où
- $head \in Goal$
- $guard \in langage$ L
- $body \in Goal$

Une PR-rule de type $h \longleftarrow g \mid$ b se lit de la façon suivante :
"Si la précondition g est vraie, alors on peut remplacer le but h par le but b"

Il existe trois cas particuliers à cette représentation :

1. La PR-rule d'initialisation : $\longleftarrow guard \mid body$
   On considère que, si une PR-rule ne possède pas de tête et que la pile des buts est vide, alors on initialise la pile avec le corps de cette PR-rule si la garde est vraie. Remarquons que ce type de Pr-rule est un cas particulier des reactive rules que nous expliciterons dans la section 5.6.3

2. La PR-rule de terminaison : $head \longleftarrow guard$
   On considère que, si une PR-rule ne possède pas de corps, alors, lorsqu'on rencontre un but correspondant à la tête de cette PR-rule et que la garde est vraie, le but sera simplement supprimé de la pile des buts. Ce qui permet à l'interpréteur de pouvoir se terminer, lorsque la pile sera vide.

3. La PR-rule d'affectation : $head \longleftarrow true \mid body$
   On considère que, si une PR-rule possède une garde toujours vraie, alors lorsqu'on rencontre un but correspondant à la tête de cette PR-rule, il est simplement remplacé par le corps de cette PR-rule. Cela correspond en quelque sorte à une affectation head := body.

Cette représentation nous permet d'expliquer aisément la manière dont une PR-rule est sélectionnée dans la base des règles pour satisfaire un but et la manière dont elle décompose ce but.

Soit
- G : le but qu'il faut satisfaire, c'est-à-dire le but qui se trouve au sommet de la pile.
- rB : la base des règles
- P : une PR-$rule \in rB$

```
TemprB = rB /*utilise une variable temporaire car la base des règles ne
             peut pas être modifiée à ce niveau ci*/
While(TemprB <> {})
  {
     Soit P  (de la forme  h <- g | b )
     Si G est unifiable avec h Alors
     {
       Si la garde g est vrai au niveau de la base des croyances alors
       {  On retire G de la pile de Buts et on y ajoute b
          Stop
       }
       Sinon
       { TemprB=TemprB \ P
       }
     Sinon
     {TemprB=TemprB \ P}
  }
```

5.6.2 Définition des PR-rules :

De manière simple et intuitive, les PR-rules peuvent être définies dans un premier temps comme ceci :

Si
- $\Pi 1, \Pi 2 \in Goal$
- $\Psi \in L$

Alors l'ensemble des PR-rules est noté PRrules et est défini inductivement par les trois assertions suivantes :

- $\Pi 1 \longleftarrow \Psi \mid \Pi 2 \in PRules$
- $\longleftarrow \Psi \mid \Pi 2 \in PRules$
- $\Pi 1 \longleftarrow \Psi \in PRules$

Néanmoins, une définition plus complète des PR-rules est nécessaire pour donner à l'agent la possibilité de raisonner sur ses plans et lui permettre de choisir quelle est la solution optimale. C'est pourquoi le concept de "*goal variables*" et le concept de "*semi-goals*" doivent être introduits pour obtenir une meilleure approche des PR-rules [20].

Les ***goal variables*** sont utilisées uniquement au niveau de la head d'une PR-rule, afin justement de permettre la révision des plans. Considérons ***Gvar*** comme l'ensemble fini de goal variables. Les ***semi-goals*** sont alors une extension de la notion de buts auxquels on rajoute le concept de Gvar.

L'***ensemble des semi-goals*** (SGoal) est donc défini par les 5 assertions suivantes :

- Gvar $\subseteq SGoal$
- Bact $\subseteq SGoal$
- Atom $\subseteq SGoal$
- Si $\alpha \in L$; Alors $\alpha? \in SGoal$
- Si $\Pi 1, \Pi 2 \in$ SGoal ; Alors $(\Pi 1; \Pi 2)$ et $(\Pi 1 + \Pi 2)$ et $(\Pi 1 || \Pi 2) \in$ SGoal

La définition la plus complète et la plus générale pour les PR-rules est donc :

Si
- $\Pi 1, \Pi 2 \in SGoal$
- $\varphi \in L$
- PRules = ensemble des PR-rules

Alors
- $\Pi 1 \longleftarrow \varphi \mid \Pi 2 \in PRules$ si toutes les occurences de toutes les goal variables de $\Pi 1$ se retrouvent dans $\Pi 2$
- $\longleftarrow \varphi \mid \Pi 2 \in PRules$ si aucune goal variable dans $\Pi 2$
- $\Pi 1 \longleftarrow \varphi \in PRules$

5.6.3 Classification des PR-rules :

Les PR-rules sont de quatre types différents [9], chaque type correspondant à une tâche particulière.

1. Failure rules

   Une failure rule a la forme suivante : $h \longleftarrow g \mid$ b

   Où
   - $h \in SGoal \setminus$ Atom
   - $g \in langage$ L
   - $b \in SGoal$

   A ce niveau, on remarque que la head de ce type de PR-rules ne peut pas contenir de buts de type achievement goal qui seront en fait traités au niveau des Plan-rules. Les failure rules permettent une replanification des plans lorsque ceux-ci échouent. Un plan est considéré comme échoué lorsque le but courant n'est plus décomposable. Deux solutions sont envisageables pour éliminer ce problème, la première est radicale et consiste à supprimer simplement le but incriminé, la deuxième plus nuancée consiste à utiliser les failure rules afin de résoudre le but grâce à d'autres alternatives telles que le "disruptor mechanism" ou le "interrupt mechanism" [17]. Le "disruptor mechanism" consiste en quelque sorte à mettre sur écoute un but particulier et à remplacer ce but par un autre lorsqu'une condition associée à se but est satisfaite. Le "interrupt mechanism" consiste non pas à remplacer le but mis sur écoute par un

autre but mais à l'interrompre en exécutant un but dit d'interruption pour lui rendre la main par après.

2. Reactive rules

   Une reactive rule a la forme suivante : $\longleftarrow g \mid$ b

   Où
   – $g \in langage$ L
   – $b \in SGoal$

   Les reactive rules ont cette caractéristique intéressante de ne plus dépendre des buts de l'agent mais bien uniquement de ses croyances. Elles peuvent, à la fois, réagir à des situations particulières et introduire de nouveaux buts. En effet, ces rules permettent de déclencher des actions spécifiques lorsque seule la condition se trouvant dans leur guard est satisfaite.

3. Plan rules

   Une Plan rule a la forme suivante : $h \longleftarrow g \mid$ b

   Où
   – $h \in Atom$
   – $g \in langage$ L
   – $b \in SGoal$

   Les plan rules déterminent les plans qui vont permettre d'atteindre les achievement goals. En effet, la plan rule suivante $h \longleftarrow g \mid$ b détermine pour l'achievement goal h un plan b.

4. Optimisation rules

   Une optimisation rule a la forme suivante : $h \longleftarrow g \mid$ b

   Où
   – $h \in SGoal \setminus$ Atom
   – $g \in langage$ L
   – $b \in SGoal$

Remarquons ici que la forme syntaxique des optimisation rules et des failure rules est identique. Pourtant, les optimisation rules ne sont pas primordiales pour le bon fonctionnement de l'agent, leur objectif est d'augmenter la performance de l'agent en remplaçant les plans d'une faible efficacité par des plans plus optimaux.

La classification des PR-rules est établie selon un ordre logique total déterminé par la priorité accordée à chaque type de PR-rules [20]. La gestion des pannes est primordiale et possède donc la priorité la plus élevée. L'optimisation des PR-rules n'est par contre pas indispensable et possède donc la priorité la plus faible.

Failure rules > Reactive rules > Plan rules > Optimisation rules

Cette classification est donc statique et de ce fait assez éloignée du comportement naturel d'un agent. L'assignation des différentes priorités entre les PR-rules devrait pouvoir être définie dynamiquement par l'agent suivant l'état de son environnement et de ses croyances. Cette nouvelle perspective octroierait à l'agent une plus grande capacité d'adaptation ainsi que la possibilité d'affiner ses choix et ses stratégies.

5.7 Les instructions impératives en 3APL

L'originalité du langage 3APL est qu'il combine à la fois de la programmation impérative et logique. Les concepts de logique du premier ordre, d'unification, de prédicat ... se retrouvent au travers des croyances, des buts, des PR-rules et des actions de base. En ce qui concerne la programmation de type impératif, le langage donne certaines possibilités qu'il faut toutefois nuancer. En effet, on retrouve au niveau de la grammaire du langage 3APL (section 5.9) les instructions Skip, If-Then-Else et While-Do.

5.7.1 Skip

SKIP

Instrucion la plus basique qui consiste à ne rien faire

5.7.2 If-Then-Else

$If\ \theta\ Then\ \Pi 1\ Else\ \Pi 2 = (\theta?; \Pi 1 + \neg\theta?; \Pi 2)$

Cette définition d'une instruction If-Then-Else en 3APL n'a pas exactement la même sémantique qu'en programmation impérative. En effet, en 3APL, la séquence de buts suivante $(\theta?; \Pi 1 + \neg\theta?; \Pi 2)$ peut, dans le cas où il n'existe aucune information sur le Test Goal $\theta?$ dans la base des croyances, ne pas s'exécuter. Le test du If-Then-Else n'est donc pas un test booléen mais un test sur la base des croyances. Remarquons ici la précédence syntaxique donnée à l'opérateur de séquence sur l'opérateur de choix non-déterministe.

5.7.3 While-do

$While\ \theta\ Do\ \Pi = (\theta?; \Pi;\ WHILE\ \theta\ DO\ \Pi + \neg\theta?; SKIP)$

Cette définition d'une instruction While-do en 3APL montre la possibilité d'utiliser la récursivité. Toutefois, comme au point précédent, le test du While-do n'est pas un test booléen, mais un test sur la base des croyances qui peut ne donner aucune réponse de type vrai ou faux.

5.7.4 Affectation

Cette instruction ne se retrouve pas dans la syntaxe du langage 3APL, pourtant, grâce au test goal, le programmeur a la possibilité d'initialiser des variables, mais absolument pas de remettre leur valeur à jour. En fait, grâce à l'opération logique d'unification se retrouvant au niveau des test goals, le programmeur peut faire des affectations caractérisées par leur unicité et leur irréversibilité.

5.8 Modélisation de l'exécution d'un agent 3APL

```
Soit gB une goalBase;
     G le but au sommet de la pile de but gB;
     P une PR-rule de la forme head <- guard | body

While gB <> {}
{
  G = élément au sommet de la pile de gB

  Si G est de "type complexe" Alors

  { Essaye de décomposer G de telle sorte que le
    premier but soit d'un type de base.
    On doit donc trouver une PR-rules P dont la head = G

    Si une telle PR-rule P (head <- guard | body) existe
       et que son guard peut être unifié à une croyance
       actuelle de l'agent
    Alors

       {G est remplacé par le body de P et la goalbase est mise
        à jour en supprimant G de la pile et en y ajoutant b.
       }
  }

  Si G est de type "action de base" Alors

    {on exécute l'action de base G en modifiant, si les
     préconditions de G sont satisfaites,  alors tous
     les beliefs dérivés des préconditions de G
     sont supprimés de la base des croyances de l'agent
     tandis que les postconditions y sont ajoutées}

  Si G est de type "test goal" Alors

    {on exécute G en liant toutes les variables non liées de G}
```

```
  Si G est de type "achievement goal" Alors

    {on essaye de trouver une PR-rules P dont la head est
     unifiable avec G

      Si une telle PR-rule P (head <- guard | body) existe
         et que son Guard peut être unifié à une croyance
         actuelle de l'agent
      Alors
         { G est remplacé par le body de P et la base des buts
           est mise à jour en supprimant G de la pile et en y
           rajoutant le body de P
         }
    }

}
```

Ce procédé continue donc jusqu'à ce que la base des buts soit vide ou que le but au sommet de la pile ne puisse plus être exécuté ou décomposé.

Exemple d'exécution de l'interpréteur 3APL :

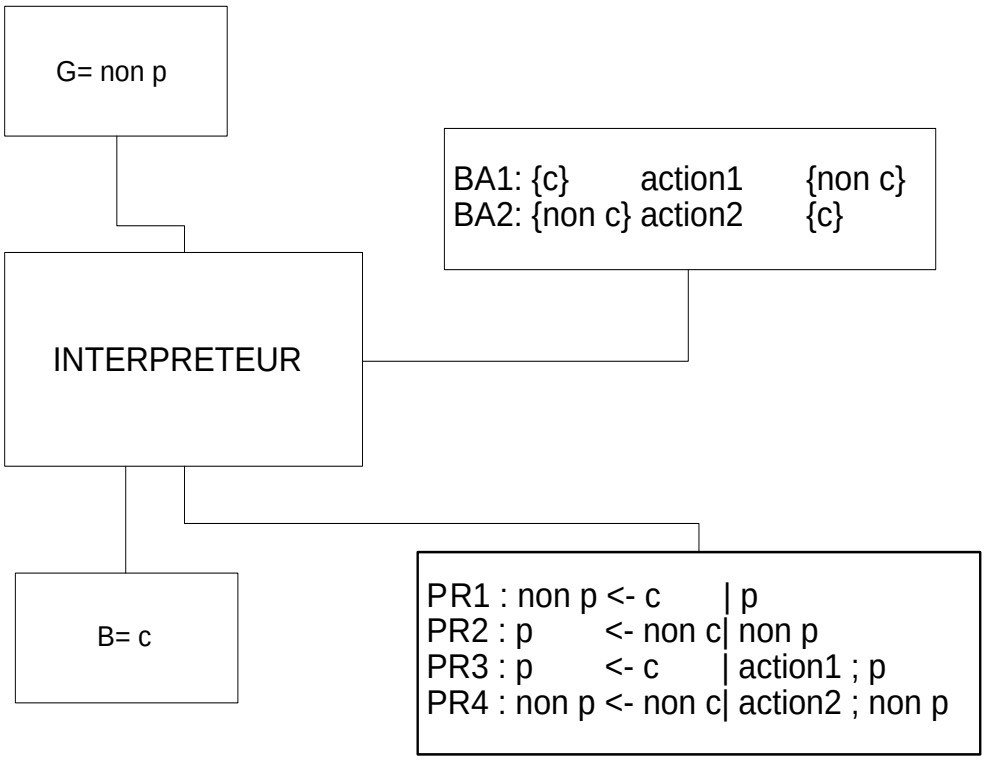


FIG. 5.3 – Exemple d'exécution de l'interpréteur 3APL

- Etape 0 : BeliefBase = c et GoalBase = non p
- Etape 1 : Applique $PR1 \Longrightarrow$ BeliefBase = c et GoalBase = p
- Etape 2 : Applique $PR3 \Longrightarrow$ BeliefBase = c et GoalBase = action1 ; p
- Etape 3 : Applique $BA1 \Longrightarrow$ BeliefBase = non c et GoalBase = p
- Etape 4 : Applique $PR2 \Longrightarrow$ BeliefBase = non c et GoalBase = non p
- Etape 5 : Applique $PR4 \Longrightarrow$ BeliefBase = non c et GoalBase = action2 ; non p
- Etape 6 : Applique $BA2 \Longrightarrow$ BeliefBase = c et GoalBase = non p
- Etape 7 : Goto Etape 1

Nous remarquons que cet exemple boucle, ce qui est une des caractéristiques les plus importante d'un langage de programmation complet. Mais, pour arriver à ce qu'un programme se termine il faut simplement utiliser des PR-rules de terminaison, c'est à dire qui ne possèdent pas de body.

5.9 La grammaire du langage 3APL

```
PROGRAM <-- CAPABILITIES ':' CAPLIST
            BELIEFBASE ':'BELIEFLIST
            GOALBASE ':' GOALLIST
            RULEBASE ':' RULELIST

CAPLIST <-- CAPLIST2 ?

CAPLIST2 <-- CAPABILITY'.'
             | CAPLIST2 ','CAPABILITY'.'

CAPABILITY <-- PRECONDITION BACTION POSTCONDITION
               | CAPABILITY

PRECONDITION <-- '{' WFFLIST '}'

POSTCONDITION <-- '{' ADD-DELETE-LIST '}'

ADD-DELETE-LIST <-- ADD-DELETE-ITEM
                    | ADD-DELETE-LIST ','
                      ADD-DELETE-ITEM

ADD-DELETE-ITEM <-- ATOM
                    | NOT ATOM

BACTION <--BACNAME '('PARLIST')'

BELIEFLIST <-- BELIEFLIST2 ?

BELIEFLIST2 <--ATOM '.'
               | BELIEFLIST2 ATOM '.'

GOALLIST <-- GOAL '.'
             |GOALLIST GOAL '.'

RULELIST <-- RULELIST2 ?

RULELIST2 <-- RULE '.'
              | RULELIST2 RULE '.'
```

```
RULE <-- HEAD ARROW GUARD '|' BODY
         | ARROW GUARD '|' BODY
         | HEAD ARROW GUARD '|'

HEAD <-- HEADSINGLE
         |HEAD ';' HEADSINGLE
         |HEAD '||' HEADSINGLE
         |HEAD '+' HEADSINGLE

HEADSINGLE <-- GOALSINGLE
               VARNAME

GUARD <-- WFF

BODY <-- GOAL

GOAL <-- GOALSINGLE
         | GOAL ';' GOALSINGLE
         | GOAL '||' GOALSINGLE
         | GOAL '+' GOALSINGLE

GOALSINGLE <-- SIMPLEGOAL
               | SKIP
               | IF WFF THEN GOALSINGLE
               | IF WFF THEN GOALSINGLE ELSE GOALSINGLE
               | WHILE WFF DO GOALSINGLE
               | BEGIN2 GOAL END

SIMPLEGOAL <-- PREDNAME '(' PARLIST ')'
               | BACTION
               | WFF '?'

WFFLIST <-- WFF
       | WFFLIST ',' WFF

WFF <-- ATOM
        | '[' WFF ']'
        | WFF AND WFF
        | WFF OR WFF
        | NOT WFF
```

```
ATOM <-- PREDNAME '(' PARLIST ')'
         | PAR2 '=' PAR
         | PAR2 '<' PAR
         | PAR2 '>' PAR

PARLIST <-- PARLIST2?

PARLIST2 <-- PAR
             | PARLIST2 ',' PAR

PAR <-- PAR2
        | PAR '-' PAR
        | PAR '+' PAR
        | PAR '*' PAR
        | PAR '/' PAR
        | FUNCTIONNAME '('PARLIST')'

PAR2 <-- NUM
         | CONSTNAME
         | VARNAME
         | '(' PAR ')'

PREDNAME <-- LOWERIDENT

FUNCTIONNAME <-- LOWERIDENT

CONSTNAME <-- LOWERIDENT

VARNAME <-- UPPERIDENT

BACNAME <-- UPPERIDENT


UPPERIDENT <-- [A-Z_][a-zA-Z0-9_]*

LOWERIDENT <-- [a-z][a-zA-Z0-9_]*
```

5.10 Exemple de programme 3APL

Voici un exemple concret de programme 3APL, écrit par Steven Anker [25], résolvant le problème de recherche et de déminage de bombes par un agent(r0) se déplaçant dans un monde constitué de trois types différents de zones : des zones contenant les bombes que l'agent doit mettre en sécurité, des zones contenant des pierres infranchissables par l'agent, et des zones libres c'est-à-dire ne contenant ni pierre ni bombe.

Le programme présenté ci-dessous ne concerne qu'un agent agissant seul dans son environnement, mais cet exemple pourrait être étendu à plusieurs agents(r0,r1,...,rn) qui seraient capables de communiquer entre eux. Ces actes de communication pouvant être simplement définis via des actions de base ou des capacités.

Par exemple un agent ayant découvert une bombe à l'emplacement de coordonnée(x,y) et l'ayant déplacée en (0,0) pourrait envoyer un message à tous les agents les plus proches de lui pour les avertir que l'emplacement de coordonnées(x,y) est sûr et qu'il ne doit donc plus être visité.

Un agent recevant un message pourra juger de la pertinence de cette nouvelle information et de la confiance qu'il peut accorder à l'expéditeur. Cette nouvelle information pourra alors être ajoutée ou non dans sa propre base des croyances.

```
CAPABILITIES:

    { robot(r0, X, Y), is(Z,X - 1), NOT stone(r0, Z, Y) }
        West()
    { NOT robot(r0,X,Y), robot(r0, X - 1, Y)}.

    { robot(r0, X, Y), is(Z,X+1), NOT stone(r0, Z, Y) }
        East()
    { NOT robot(r0,X,Y), robot(r0, X + 1, Y) }.

    { robot(r0, X, Y), is(Z,Y - 1), NOT stone(r0, X,Z) }
        North()
    { NOT robot(r0,X,Y), robot(r0, X, Y - 1) }.


    { robot(r0, X, Y), is(Z,Y+1), NOT stone(r0, X, Z) }
        South()
    { NOT robot(r0,X,Y), robot(r0, X, Y+1) }.
```

```
    { bomb(r0, X, Y), robot(r0, X, Y)}
        Pickup()
    { carriesBomb(r0), NOT bomb(r0, X, Y) }.

    { carriesBomb(r0), robot(r0, X, Y) }
        Drop()
    { NOT carriesBomb(r0), bomb(r0, X, Y) }.

BELIEFBASE:

    robot(r0, 5, 5).

GOALBASE:

    cleanupBombs().

RULEBASE:

    // Les quatres règles suivantes sont utilisées pour réviser
    // les goals de l'agent quand il est bloqué par une pierre.

    South(); findbomb(r0, V,W)
     <- robot(r0, X, Y) AND is(Z,Y + 1) AND stone(r0, X, Z) |
        IF [NOT stone(r0, X - 1, Y)] AND [X > 0] THEN
            BEGIN
                West();
                South();
                findbomb(r0, V,W)
            END
        ELSE SKIP.

    East(); findbomb(r0, V,W) <- robot(r0, X, Y) AND is(Z,X + 1)
                                                 AND stone(r0, Z, Y) |
        IF [NOT stone(r0, X, Y - 1)] AND [Y > 0] THEN
            BEGIN
                South();
                East();
                findbomb(r0, V,W)
            END
        ELSE SKIP.
```

```
North(); findbomb(r0, V,W)
 <- robot(r0, X, Y) AND is(Z,Y - 1) AND stone(r0, X, Z) |
    IF [NOT stone(r0, X + 1, Y)]  THEN
        BEGIN
            East();
            North();
            findbomb(r0, V,W)
        END
    ELSE SKIP.

West(); findbomb(r0, V,W) <- robot(r0, X, Y) AND is(Z,X - 1)
                                              AND stone(r0, Z, Y) |
    IF [NOT stone(r0, X, Y + 1)] THEN
        BEGIN
            North();
            West();
            findbomb(r0, V,W)
        END
    ELSE SKIP.

// Les deux règles suivantes sont utilisées pour réviser
// les goals de l'agent si la bombe qu'il cherchait a disparue.

findbomb(r0, X,Y) <- [NOT bomb(r0, X,Y)] AND nearestBomb(r0, U,V) |
    findbomb(r0, U,V).

bomb(r0, X, Y)? <- [NOT bomb(r0, X,Y)] AND nearestBomb(r0, U,V) |
    findbomb(r0, U,V).


goto(X, Y) <- robot(r0, X0, Y0) |
    IF X = X0 AND Y = Y0 THEN SKIP
    ELSE
        BEGIN
            IF X < X0 THEN West()
            ELSE IF X0 < X THEN East()
            ELSE IF Y < Y0 THEN North()
            ELSE IF Y0 < Y THEN South();
            goto(X, Y)
        END.
```

```
findbomb(r0, X, Y) <- robot(r0, X0, Y0) AND bomb(r0, X,Y) |
    IF X = X0 AND Y = Y0 THEN SKIP
    ELSE
        BEGIN
            IF X < X0 THEN West()
            ELSE IF X0 < X THEN East()
            ELSE IF Y < Y0 THEN North()
            ELSE IF Y0 < Y THEN South();
            findbomb(r0, X, Y)
        END.

cleanupBombs() <- nearestBomb(r0, X, Y) |
    findbomb(r0, X, Y);
    bomb(r0, X, Y)?;
    Pickup();
    goto(0, 0);
    Drop();
    cleanupBombs().
```

En plus de ce programme 3APL il est nécessaire d'utiliser un programme prolog pour définir la règle nearestBomb(r0,U,V) qui détermine l'emplacement (U,V) de la bombe la plus proche du robot r0. En effet, il est clair que la base des croyances ne peut pas contenir un fait déterminant qu'elle est la bombe le plus proche d'un robot puisque celui-ci se déplace en permanence. En bref, le prédicat prolog nearestBomb est constitué de deux règles ; **bombs** qui fournit deux listes contenant respectivement les positions X et Y des bombes et **minimumBomb** qui donne comme résultat la bombe qui se trouve à la plus petite distance d'un robot.

```
table :-
   nearestBomb/3, miminumBomb/6,
   smaller/6, bombs/2, bomb/3,
   robot/3, stone/3.

:- dynamic stone/3. :- dynamic bomb/3.

nearestBomb(R, X,Y) :-
   bombs(R,L1, L2), robot(R,V,W),
   minimumBomb(L1, L2, V, W, X, Y).
```

```
bombs(R,L1, L2) :-
   findall(X, bomb(R,X,_), L1),
   findall(Y, bomb(R,_,Y), L2).

minimumBomb([H1], [H2],_,_, X, Y) :- X = H1, Y = H2.

minimumBomb([H1|L1], [H2|L2], V, W, X, Y):-
    minimumBomb(L1, L2, V, W, P, Q),
    (smaller(V, W, P, Q, H1, H2) -> (X = H1, Y = H2) ; (X = P, Y = Q)).

abs(X,X) :- not(X < 0).
abs(X,Y) :- X < 0, Y is -X.

smaller(V, W, X, Y, H1, H2) :-
    P1 is (X - V),  abs(P1,Pa1),
    P2 is (Y - W),  abs(P2,Pa2),
    Q1 is (H1 - V), abs(Q1,Qa1),
    Q2 is (H2 - W), abs(Q2,Qa2),
    (Qa1 + Qa2) < (Pa1 + Pa2).
```

5.11 Implémentation du langage 3APL

5.11.1 Présentation du projet

Après la publication de la thèse de Koen Hindriks, un groupe de recherche s'est formé à l'université d'Utrecht pour fournir une implémentetion de 3APL en C++. L'étape actuelle de ce projet est de réaliser une implementation de ce langage sous la forme d'un package JAVA utilisant un moteur XSB-prolog. Dans le cadre de notre stage nous avons intégré cette équipe de recherche composée de messieurs Mehdi Dastani et Meindert Kroese ainsi que du Professeur Frank de Boer. L'objectif qui nous a été assigné consistait à réaliser l'implémentation de la base des croyances et des interactions minimales entre agents. Une documentation complète en javadoc nous était aussi demandée afin que notre code soit facilement réutilisable. Le code et la javadoc associés à cette implémentation se trouvent sur le cd-rom en annexe de ce mémoire. Le progrès majeur par rapport à la version précédente est lié au fait que chaque agent possède désormais sa propre base des croyances indépendante. On oublie donc ici le concept de tableau noir ou de base des croyances centralisée pour tous les agents. Toutefois ce projet n'en était qu'à ses débuts. Ainsi, avant de pouvoir réellement entrer dans cette phase d'implémentation, il a fallu réfléchir à une architecture générale pour l'implémentation de l'agent 3APL.

5.11.2 Architecture générale

Comme nous l'avons dit précédemment, l'implémentation du langage 3APL en JAVA consiste à créer un package fournissant des classes permettant de programmer des agents. L'idée principale de ce package est de faciliter au maximum la vie du programmeur en ne l'autorisant à accéder qu'au seul constructeur de la classe *Agent*. L'intérêt est de permettre à l'utilisateur d'implémenter des agents en ayant à se préoccuper, quasi uniquement, que de la spécification 3APL de ses agents. En effet, le constructeur de cette classe prend comme paramètres un nom sous forme de string qui identifie l'agent, un registre qui constitue en quelque sorte un annuaire téléphonique que les agents consultent pour pouvoir communiquer entre eux et enfin, la beliefBase, la goalBase, la ruleBase et les capabilities de l'agent, sous forme de fichiers.

La classe *Agent* étend la classe *Thread*, permettant ainsi l'exécution simultanée et concurrente de plusieurs agents. L'interpréteur 3APL devra être implémenté à l'intérieur de la méthode *run()*, exécutée au lancement du thread.

En pratique, un programmeur voulant implémenter plusieurs agents créera une classe Java dont la méthode *main()* déclare les agents de la manière suivante :

```
public static void main(){

Agent a1 = new Agent(NameofAgentA1, fileOfBeliefsA1,
fileOfGoalsA1, fileOfPRrulesA1, fileOfCapabilitiesA1, Registry)

Agent a2 = new Agent(NameofAgentA2, fileOfBeliefsA2,
fileOfGoalsA2, fileOfPRrulesA2, fileOfCapabilitiesA2, Registry)

... }
```

Toutefois, la classe *Agent* n'est pas la seule classe du package 3APL. D'autres classes, telles que *BeliefBase*, *GoalBase*, *RuleBase*, ... sont utilisées par la classe *Agent* mais l'utilisateur n'en a pas connaissance. La figure 5.4 illustre l'architecture générale du package 3APL. Le prochaine section s'intéressera plus spécifiquement à la classe *BeliefBase*, que nous avons implémentée durant notre stage à l'Université d'Utrecht.

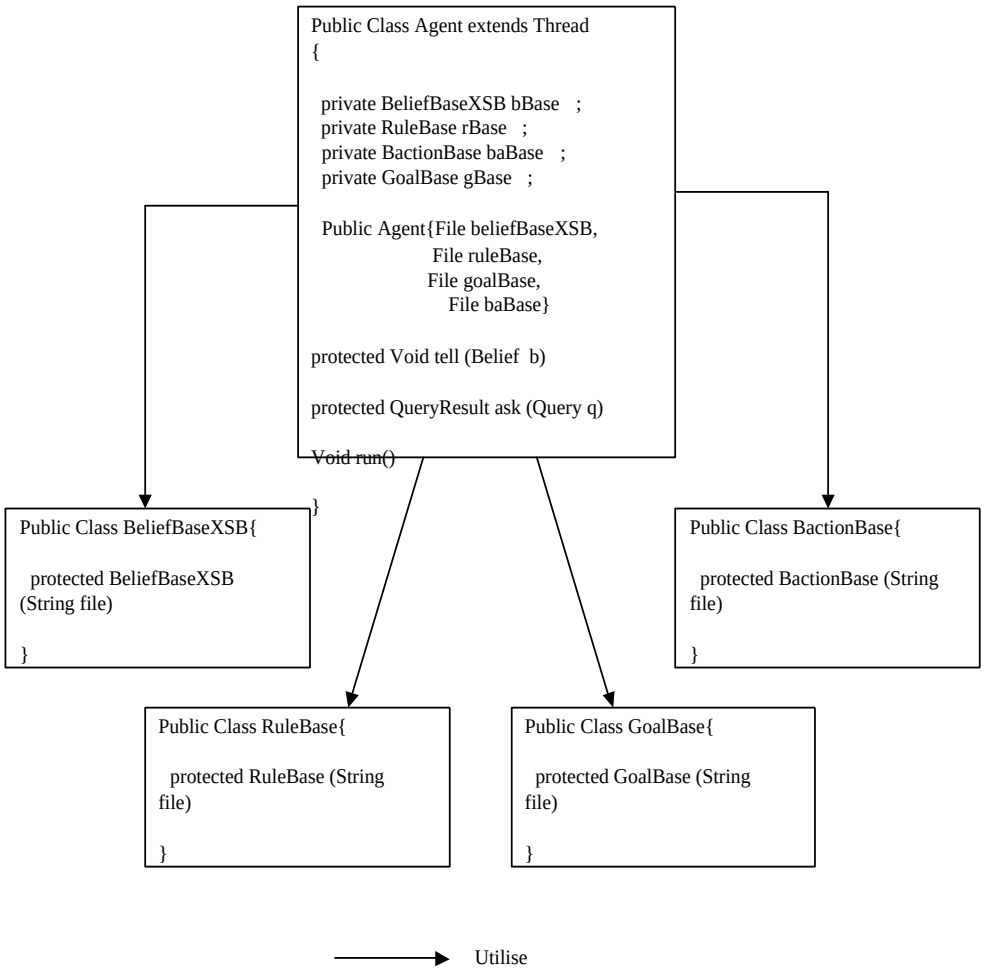


FIG. 5.4 – Architecture générale du package 3APL

5.11.3 Implémentation de la base des croyances 3APL

La classe *BeliefBase* fournit trois méthodes du type protected permettant d'interroger la base des croyances, d'y ajouter et d'y retirer une croyance. Elle dispose aussi d'un constructeur, permettant d'initialiser la base des croyances à partir d'un fichier. Afin de pouvoir implémenter ces méthodes, la classe BeliefBase utilise d'autres classes présentées dans la figure 5.5.

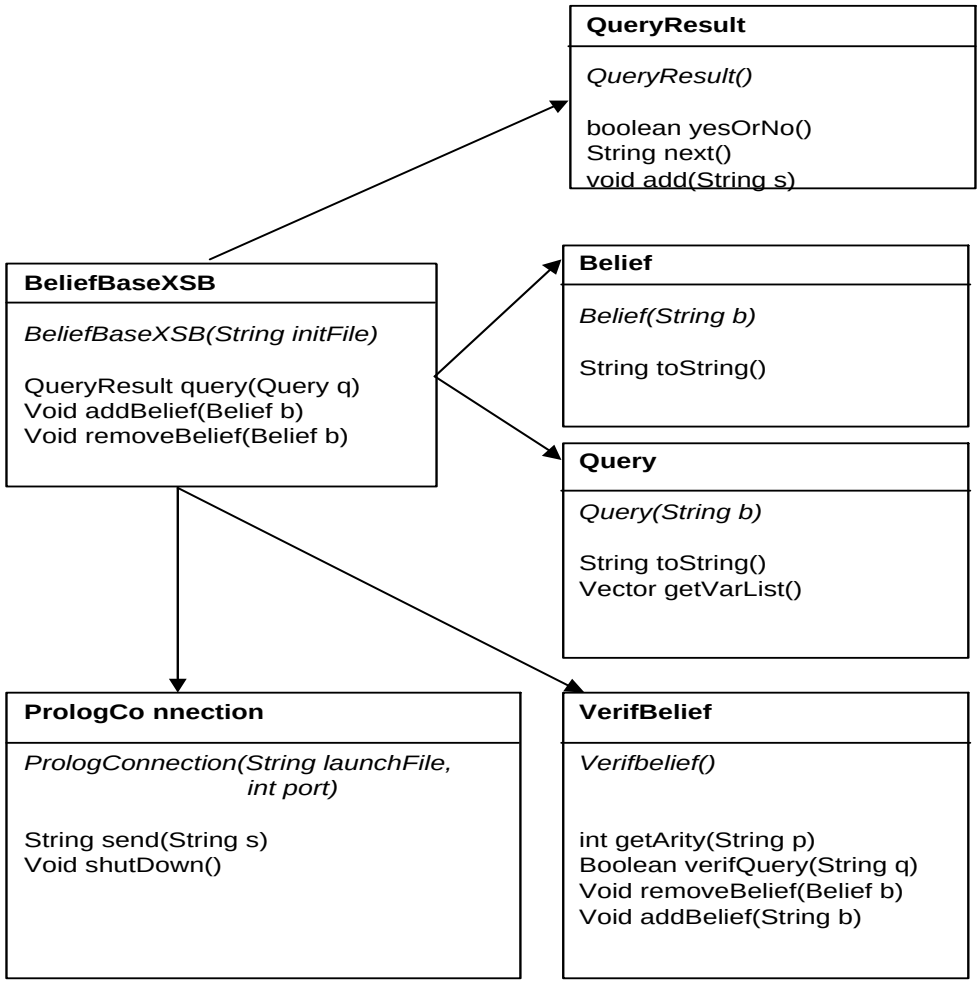


FIG. 5.5 – Architecture de la base des croyances

La méthode $QueryResult\ query(Query\ q)$ permet ainsi d'interroger la base des croyances (consistant en un ensemble de prédicats prolog) avec un atome prolog. La méthode retourne un objet $QueryResult$ contenant la réponse de prolog (yes/no) et toutes les substitutions. Notons également que la méthode $QueryResult\ query(Query\ q)$ prend en entrée un objet de type $Query$, et non un string. Ceci permet en fait de vérifier la syntaxe de la requête (dans le constructeur de la classe Query) avant de l'envoyer au moteur prolog.

Les méthodes $void\ addBelief(Belief\ b)$ et $void\ removeBelief(Belief\ b)$ ajoutent ou retirent une croyance de la base des croyances. Une croyance consiste en n'importe quelle formule prolog. Nous utilisons également un objet $Belief$, pour la même raison que celle évoquée ci-dessus.

La classe $BeliefBase$ utilise la classe $PrologConnection$, qui constitue en tant que tel le moteur XSB-prolog avec lequel Java communique. Ainsi, le fait qu'un processus XSB-prolog externe tourne est caché à l'utilisateur de la classe $BeliefBase$. La figure suivante illustre la communication entre JAVA et XSB-prolog.

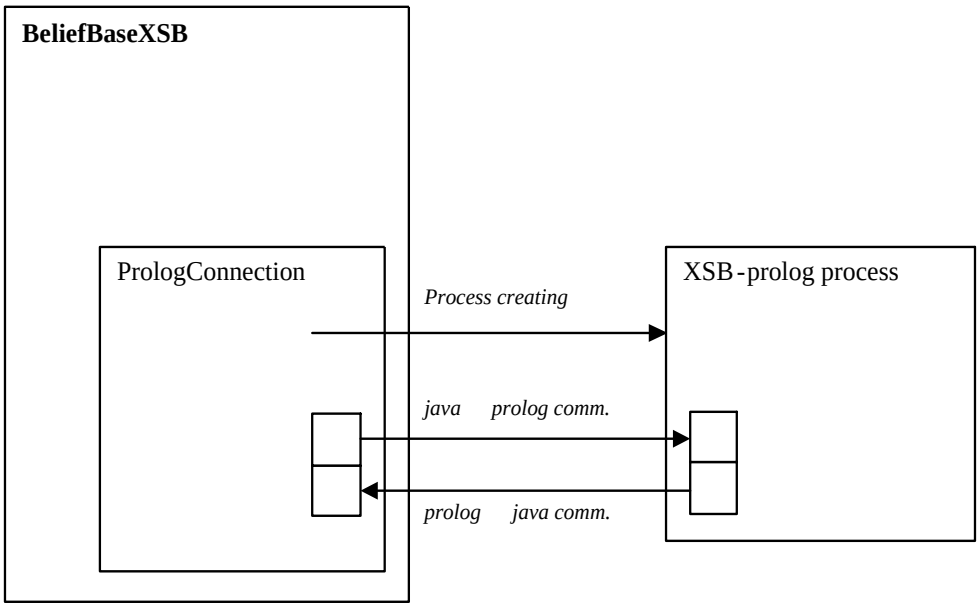


FIG. 5.6 – Communication entre java et XSB-prolog

Cette communication se fait au moyen de sockets. Ainsi, le processus XSB-Prolog joue le rôle du serveur, et la classe $PrologConnection$ joue le rôle du client. Ici se justifie l'utilisation des objets $Belief$ et $Query$, permettant de vérifier la syntaxe des formules avant de les envoyer à XSB-prolog, car il s'est révélé qu'une erreur de syntaxe au niveau de XSB-prolog coupe le canal de communication.

Nous avons également implémenté l'objet $VerifBelief$, qui tient à jour la liste des prédicats dont dispose le moteur XSB-prolog, et détermine si une requête (query) est manifestement fausse avant de l'envoyer au moteur. Cette objet est en fait une simple table de hashing qui contient les différents prédicats contenus dans le moteur. On peut la représenter de la façon suivante : $Hashmap < nameofpredicate, listof < arity, numberofoccurences >>$. L'ensemble des clés de cette table de hashing contient les noms des prédicats et à chacune de ces clés correspond une liste de couples composés d'une part de l'arité du prédicat et d'autre part du nombre d'occurrences de ce prédicat

avec cette même arité. On considère donc qu'une requête est acceptable si son nom appartient à l'ensemble des clés de la table et si l'arité se retrouve dans la liste des couples. Le nombre d'occurrences n'est en fait utilisé que lorsqu'on ajoute ou retire des prédicats dans le moteur XSB-prolog afin de maintenir la table à jour.

Tester la base des croyances 3APL

Pour tester notre classe BeliefBase nous avons simplement créé une pseudo classe Agent dont le constructeur ne contenait comme paramètre que le fichier de la base des croyances. Voici donc un exemple simple de test sur la base des croyances.

```
Class Agent {
    BeliefBaseXSB bBase;

    Public Agent(String fileOfBeliefs)
    {
        bBase = new BeliefBaseXSB(fileOfBeliefs);

       ...
    }

    public void run()
    {
        //Interpreter loop

        ...

        // Adding a belief to the BeliefBase
        bBase.addBelief(  new Belief("p(X,Y).")  );

        ...

        // Removing a belief from the BeliefBase
        bBase.removeBelief(  new Belief("p(X,Y).")  );

        ...

        // Querying the BeliefBase
```

```
        Query q  =  new Query("p(X,b).");
        QueryResult result = bBase.query( q );

        While (result.yesOrNo() = true) //while there is more results
        {
            System.out.println( result.next() );
        }

        ...
    }
```

5.11.4 Implémentation des interactions entre agents

La communication one-to-one entre agents 3APL ne nécessite dans un premier temps que deux fonctionnalités : l'agent doit pouvoir d'une part, dire à un autre agent qu'il croit la croyance b et d'autre part pouvoir interroger la base des croyances d'un autre agent en formulant des requêtes. Cette élaboration des moyens de communication entre agents 3APL devait être la plus basique et la plus simple possible. L'idée étant de pouvoir dans un avenir proche passer à un mode de communication basé sur le standard défini par FIPA ACL.

Les méthodes de communication

Chaque agent est doté de deux méthodes liées à ses capacités de communication : ask et tell. Celles-ci sont simplement implémentées dans la classe agent. Il faut garder à l'esprit que ces deux méthodes sont du type protected, c'est-à-dire que l'utilisateur du package 3APL n'en a pas connaissance. Ces méthodes ne seront nécessaires que dans la partie implémentation de l'interpreteur 3APL. Le programmeur voulant faire communiquer ses agents devra utiliser les actions de base et les PR-rules. La méthode '***ask(Query q)***' renvoie un objet QueryResult qui correspond donc à la réponse de la requête q, c'est-à-dire simplement la réponse que donne la base des croyances lorsqu'on appelle la méthode "beliefBase.query(q)". La méthode ***'tell(Belief b)'*** renvoie void et ajoute la croyance b à la base des croyances de l'objet sur lequel cette méthode est appliquée grâce à la méthode "beliefBase.addBelief(belief)" sans examen de la crédibilité de l'agent qui envoie le Belief b.

Le registre

De plus, il est nécessaire d'utiliser un registre dans lequel tous les agents s'enregistrent dès leur création. Les agents ne connaissent en fait que le nom des autres agents appartenant à leur environnement. Il est à noter que nous avons pris comme hypothèse simplificatrice que chaque agent possède un nom unique et identique pour tous les autres agents afin d'éviter tous problèmes liés à l'ontologie. Cet objet "registry" est passé comme paramètre dans le constructeur de la classe agent. La fonction première de ce registre étant de permettre à un agent quelconque de trouver à partir du nom d'un agent, l'instance de la classe agent qui correspond à ce nom. Sur cette instance, l'agent pourra alors simplement appeler les méthodes tell et ask.

En pratique, le registre est représenté par une table de hashing qui contient le nom identifiant donné à l'agent et l'objet agent correspondant. Plusieurs méthodes différentes peuvent être appliquées sur cet objet "registry". Des méthodes comme **addAgent(String name, Agent a)** et **removeAgent(String name)** qui respectivement ajoute et retire un agent au registre. Ou encore d'autres méthodes comme **getAgent(String name)** qui renvoie l'objet agent correspondant à l'agent de nom name, **getAllAgentNames()** qui renvoie un set contenant tous les noms des agents appartenant au registre, **isAgentIn(String Name)** qui renvoie un booléen à vrai si Name est le nom d'un agent appartenant au registre et faux sinon.

Tester les moyens de communication 3APL

Voici en pratique la façon, dont un agent nommé agent1 va envoyer à l'agent2 un message du type "tell(p(a,X))." qui signifie qu'il croit la croyance "p(a,X)." en supposant l'agent1 honnête.

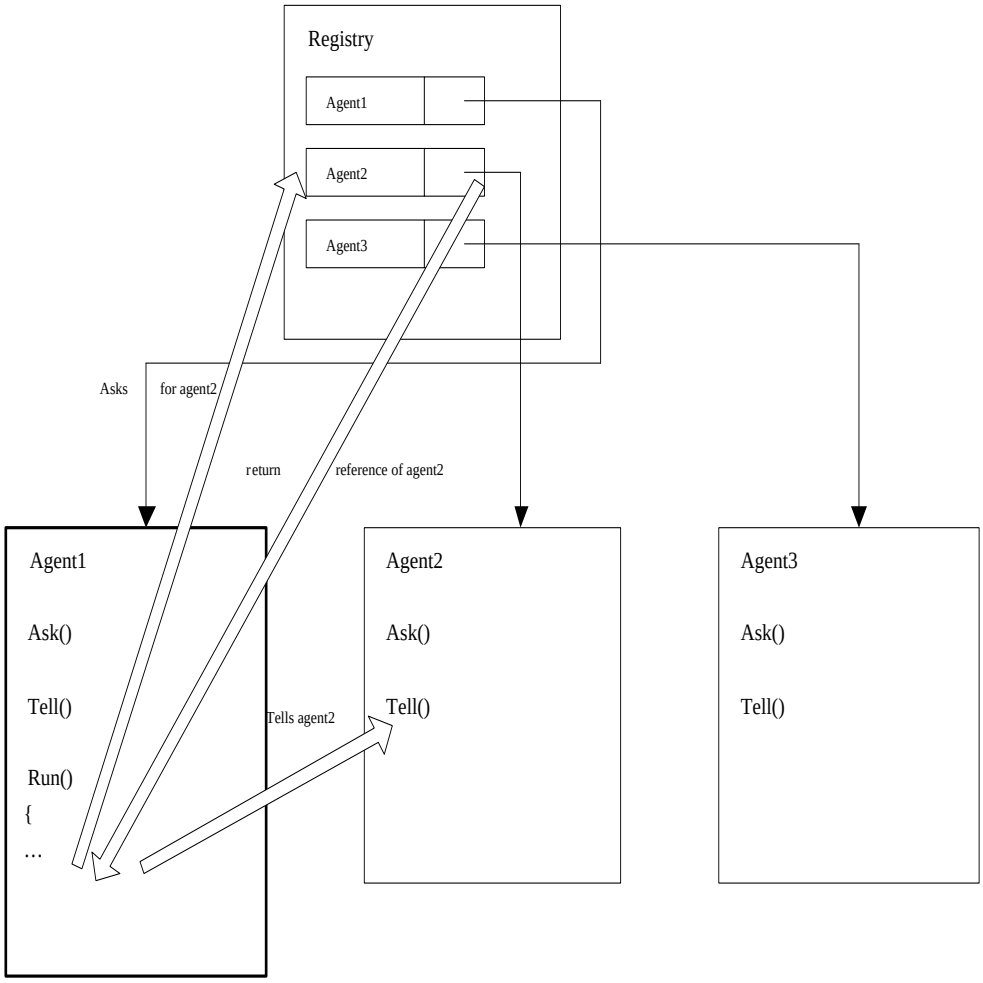


FIG. 5.7 – Communication entre agents 3APL

Notons toutefois que, comme cela a déjà été annoncé, les méthodes de communication ne peuvent pas être appelées par l'utilisateur du package 3APL. Ces méthodes seront invoquées en interne dans la méthode run() qui correspond à l'interpréteur de l'agent. Comme cette partie n'est pas encore implémentée nous avons donc trouvé une solution simple en rendant ces méthodes publiques juste durant notre phase de test.

Voici la méthode run de l'agent1 :

```
Public void run() {
    Agent  a2 = registry.getAgent("agent2");
    a2.tell( new Belief("p(a,X)."));
    ... }
```

5.11.5 Commentaires

Nous avons été confrontés durant cette implémentation à différents problèmes techniques liés notamment à l'interaction entre JAVA et XSB-prolog. Nos problèmes ont été essentiellement dus au manque de documentation du langage XSB-prolog surtout au niveau des messages renvoyés par les processus XSB-prolog. XSB-prolog nous a été recommandé par l'équipe de recherche car ce langage avait été utilisé dans l'élaboration de la version C++ de 3APL. En effet, XSB-prolog contient une interface d'interaction avec C++ déjà implémentée et facile d'utilisation, mais ce même type d'interface n'existe pas encore avec JAVA. Pour faire communiquer JAVA et XSB-prolog nous avons donc choisi l'utilisation des sockets. Notre argument principal était basé sur un critère de réutilisabilité. L'idée étant que si on voulait changer de version de prolog, du moment que cette version supportait les sockets, il suffisait simplement de modifier la classe prologconnection et le serveur prolog.

De plus, l'interaction entre XSB-prolog et JAVA est assez fragile en cours d'exécution car toutes erreurs même syntaxiques engendrées dans le moteur XSB-prolog coupent le canal de transmission. C'est pourquoi nous avons été obligés de créer les classes $Query$, $Belief$, $Verifbelief$ qui vérifient les requêtes et les croyances envoyées à XSB-prolog. Cette vérification engendre un manque d'efficacité car elle fait double emploi avec le moteur XSB-prolog qui remplit déjà ces fonctionnalités. Ces erreurs entraînent dans tous les cas l'arrêt du système, mais avec nos méthodes de vérification le programmeur peut obtenir des indications claires sur l'erreur commise.

Il très facile en cours d'exécution d'ajouter un nouveau prédicat dans le moteur XSB-prolog. Toutefois, enlever un prédicat au moteur XSB-prolog en cours d'exécution s'est révélé plus difficile, car la seule solution était de suspendre tous les accès à la base des croyances, de couper le moteur XSB-prolog et de le relancer la base des croyances avec un nouveau fichier de croyances tenu à jour grâce à la classe verifbelief qui maintient une liste de tous les prédicats contenu dans la base des croyances. Cette méthode est peu efficace, mais d'après l'équipe de recherche le nombre de suppressions de croyances est négligeable dans l'exécution de l'interpréteur 3APL et donc ce manque d'efficacité est dans une certaine mesure acceptable.

Suite à ces problèmes, nous avons songé à la possibilité d'utiliser des modules JAVA qui "simulent" Prolog. Cette option, nous a paru fort peu efficace et inadéquate par rapport à la philosophie 3APL.

En ce qui concerne les méthodes de communication entre agents, elles peuvent difficilement être plus simples. Une première évolution nous paraît pourtant envisageable et nécessaire. En effet, lorsqu'un *agent x* appelle par

exemple la méthode tell(belief b) de l'*agent y*, l'*agent y* n'a pas le choix et doit ajouter cette croyance b à sa base des croyances sans aucune nuance. L'idée serait simplement de considérer ces méthodes de communication comme un senseur à part entière de l'agent. A la place d'ajouter la croyance b à la base des croyances de l'*agent y*, il serait plus pertinent de rajouter une croyance du type "receive(agent x,tell(b))." . L'*agent y* pourrait donc adopter une politique différente grâce à ses PR-rules suivant l'agent qui envoie une croyance ou qui fait une requête.

Chapitre 6

Le langage de programmation AgentSpeak(L)

Le langage de programmation AgentSpeak(L) a été conçu par Anand S. Rao [40] et est basé sur le Procedural Reasoning System (PRS) et sur Distributed Multi-Agent System (dMARS) [10]. Ce langage orienté BDI suppose donc que les agents sont en interaction continue avec leur environnement et qu'ils sont dotés de trois attitudes mentales : les croyances, les désires et les intentions.

Ce chapitre se décomposera en trois parties, la première introduira la spécification d'un agent AgentSpeak(L), la deuxième expliquera le fonctionnement de ce même type d'agent et enfin nous présenterons une comparaison entre l'agent AgentSpeak(L) et l'agent 3APL.

6.1 Spécification d'un agent AgentSpeak(L)

Un agent AgentSpeak(L) est spécifié à l'aide de cinq ensembles [40] : l'ensemble de ses actions, de ses croyances (beliefs), de ses plans, de ses intentions et des événements (events). Ces agents grâce à leurs senseurs, qui interagissent avec leur environnement, modifient leurs croyances et leurs buts. Suite à ces changements de croyances et/ou de buts, les agents choisissent des plans qui vont engendrer des intentions.

Afin de correspondre à l'architecture présentée dans la figure 1.1, l'architecture d'un agent AgentSpeak(L) peut être schématisée de la manière présentée dans la figure 6.1.

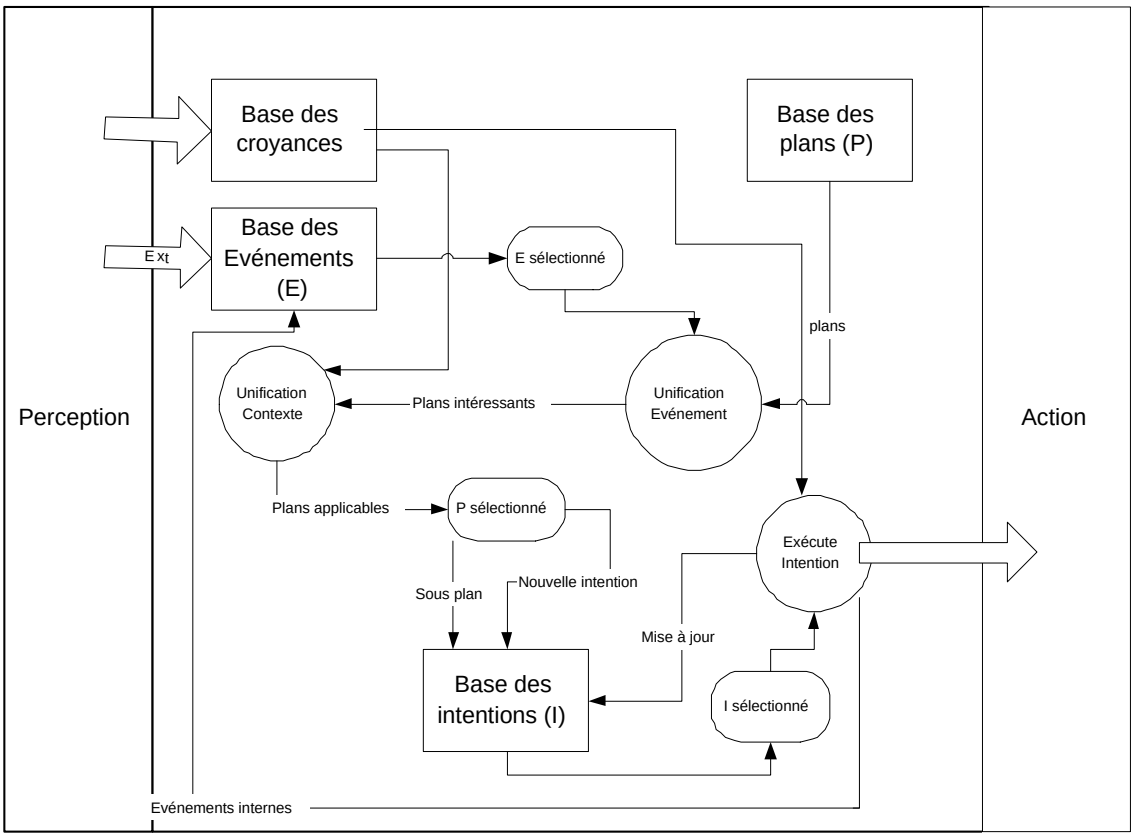


FIG. 6.1 – Architecture d'un agent AgentSpeak(L)

Nous allons dans la suite expliciter ce schéma. Mais, dans un premier temps nous, allons aborder sept concepts fondamentaux qui sont étroitement liés à ce type d'agent : les actions, les croyances, les buts, les événements déclenchant (triggering events), les plans, les intentions et les événements [40].

6.1.1 Les actions

Les *actions* déterminées pour l'agent lui permettent de modifier son environnement. L'exécution de ces actions dépend principalement des buts de l'agent et de l'input qu'il reçoit de son environnement.

On définit donc une *action* comme :

Si
- a est le symbole d'une action
- t1,...,tn sont des termes du premier ordre

Alors
- a(t1,...tn) est une action

6.1.2 Les croyances

Les *croyances* d'un agent AgentSpeak(L) représentent les faits considérés comme vrais à un moment donné par l'agent. Ces croyances portent non seulement sur l'agent lui-même, mais aussi sur son environnement et sur d'autres agents.

La définition d'une *croyance* est donc la suivante :

Si
- Pred = ensemble des prédicats
- Term = ensemble des termes

Alors
- atomes de croyance : At = P(t1,...,tn) | P $\in$ *Pred* d'arité n et t1,...,tn $\in T$
- litéraux : $Lit = At \cup \neg At$
- croyances : c = Lit

6.1.3 Les buts

Les *buts* d'AgentSpeak(L) définissent l'état dans lequel l'agent veut arriver et sont de deux types différents ; d'une part, un **Achievement Goal** noté $!g(t)$ indique que l'agent veut rendre g(t) vraie et d'autre part, un **Test Goal** noté $?g(t)$ montre que l'agent veut tester si la formule g(t) est vraie ou non.

La définition d'un *but* est donc la suivante :

Si
- g est un prédicat
- t1,...tn sont des terms

Alors
- !g(t1,...,tn) et ?g(t1,...,tn) sont des buts

6.1.4 Les événements déclenchant

Les *événements déclenchant (triggering events)* d'AgentSpeak(L) sont de quatre types différents ; c'est-à-dire que leur type est déterminé suivant le rôle qu'ils jouent, soit ajouter ou supprimer des croyances soit ajouter ou supprimer des buts. Ces événements déclenchant permettent à l'agent de modifier ses croyances et/ou ses buts lorsqu'il détecte une modification de son environnement ou lorsqu'il acquiert un nouveau but.

La définition d'un *événement déclenchant* est donc la suivante :

Si
- b(t) est une croyance
- ?g(t) et !g(t) sont des buts
- + = opérateur d'addition de buts et de croyances
- - = opérateur de suppression de buts et de croyances

Alors
- +b(t), +!g(t), +?g(t1), -b(t), -!g(t), -?g(t1) sont des événements déclenchant

6.1.5 Les plans

Les *plans* d'AgentSpeak(L) spécifient la manière dont l'agent va réaliser ses objectifs. Une caractéristique intéressante de ces plans est qu'ils sont "context-sensitives" car ils dépendent des croyances de l'agent et donc aussi de son environnement.
Ces plans sont représentés de la manière suivante :

événement déclenchant : $Contexte \leftarrow corps$

Où
- l'événement déclenchant détermine la condition de déclenchement de ce plan.
- Le contexte est une suite de croyances qui doivent être satisfaites pour que l'agent puisse exécuter ce plan.
- Le corps est une séquence de buts et/ou d'actions qui spécifie les buts que l'agent doit atteindre et les actions qu'il doit exécuter.

Un *plan* est donc défini comme suit :

Si
- e est un événement déclenchant
- b1,...,bm sont des croyances
- h1,...,hn sont des buts ou des actions

Alors
- $e : b1 \wedge ... \wedge bm \leftarrow h1; ...; hn$ est un plan

6.1.6 Les intentions

Les *intentions* d'AgentSpeak(L) définissent l'ensemble des plans qui ont été choisis par l'agent pour arriver à satisfaire ses (sous)buts. Lors de l'exécution de l'agent ce seront uniquement les plans des intentions appartenant à cette ensemble qui seront exécutés. Les intentions sont construites durant l'exécution de l'agent et chacune d'elle forme une séquence ou une pile de plans dits partiellement instanciés.

Une *intention* est donc définie comme suit :

Si
- p1,...,pn sont des plans

Alors
- $[p1, ..., pn]$ est une intention ou p1 est la base de la pile et pn le sommet de la pile

6.1.7 Les événements

Les *événements* d'AgentSpeak(L) sont des couples événement déclenchant et intention de la forme (e,i). Ces événements déterminent donc quelle intention est nécessaire afin que l'événement déclenchant associé se déclenche. Il existe deux types différents d'événement ; d'une part, les **événements externes** qui ont comme caractéristique principale d'être de la forme (e,True) ce qui implique que l'événement déclenchant sera toujours déclenché et d'autre part, les autres événements qui sont appelés **événements internes**.

Un *événement* se définit donc comme suit :
Si
- Et = ensemble des événements déclenchant
- I = ensemble des intentions
- $e \in Et$
- $i \in I$

Alors
- (e,i) est un événement

6.1.8 Exemple : Robot ramasseur de déchets

Pour illustrer ces différentes notions voici un exemple extrêmement simple [40] concernant un robot dont le but est de ramasser des déchets et de les déposer dans une poubelle. Cet agent est doté de trois actions différentes ; premièrement il a la possibilité de ramasser des déchets (pick), deuxièmement il est capable de se déplacer dans l'espace (move) et troisièmement il peut déposer les déchets dans une poubelle (drop).

Si on considère que les croyances de l'agent sont de cet ordre :

- adjacent(a,b).
- adjacent(b,c).
- adjacent(c,d).
- location(robot,a).
- location(waste,b).
- location(bin,d).

Trois plans ont été envisagés pour cet agent ; les deux premiers plans consistent à permettre à l'agent de se déplacer à une position déterminée et le troisième permet à l'agent de ramasser et de déposer un déchet dans

une poubelle lorsqu'il se trouve sur la position exacte du déchet. Les deux premiers plans sont liés à des événements déclenchant du type ajout d'un but tandis que le dernier est lié à un événement déclenchant du type ajout d'une croyance.

1. +!location(robot,X) :location(robot,X) ← true.
2. +!location(robot,X) :location(robot,Y) ∧ (not (X=Y)) ∧
   adjacent(Y,Z) ←
   move(Y,Z) ; +!location(robot,X).
3. +location(waste,X) :location(robot,X) ∧ location(bin,Y) ←
   pick(waste) ; !location(robot,Y) ;drop(waste).

Supposons l'apparition d'un événement externe dont l'événement déclenchant est +!location(robot,b). Si on s'intéresse à l'ensemble des croyances, il est évident que seul le deuxième plan est applicable avec comme unificateur $[X/b, Y/a, Z/b]$. Ce nouvel événement va donc engendrer une nouvelle intention (ne contenant dans sa pile qu'un seul plan) de la forme :

[+!location(robot,b) :location(robot,a) ∧ (not (b=a)) ∧
adjacent(a,b) ←
move(a,b) ; +!location(robot,b).]

6.2 Fonctionnement d'un agent AgentSpeak(L)

Le fonctionnement classique d'un agent AgentSpeak(L) se déroule en sept étapes [8]. La figure suivante présente les quatre premières étapes qui ont pour rôle de mettre à jour l'ensemble des intentions de l'agent. Les trois dernières étapes décrivent l'exécution des intentions par l'agent. La figure représentant l'architecture d'un agent AgentSpeak(L) présente de manière globale les principes que nous allons développer dans la suite.

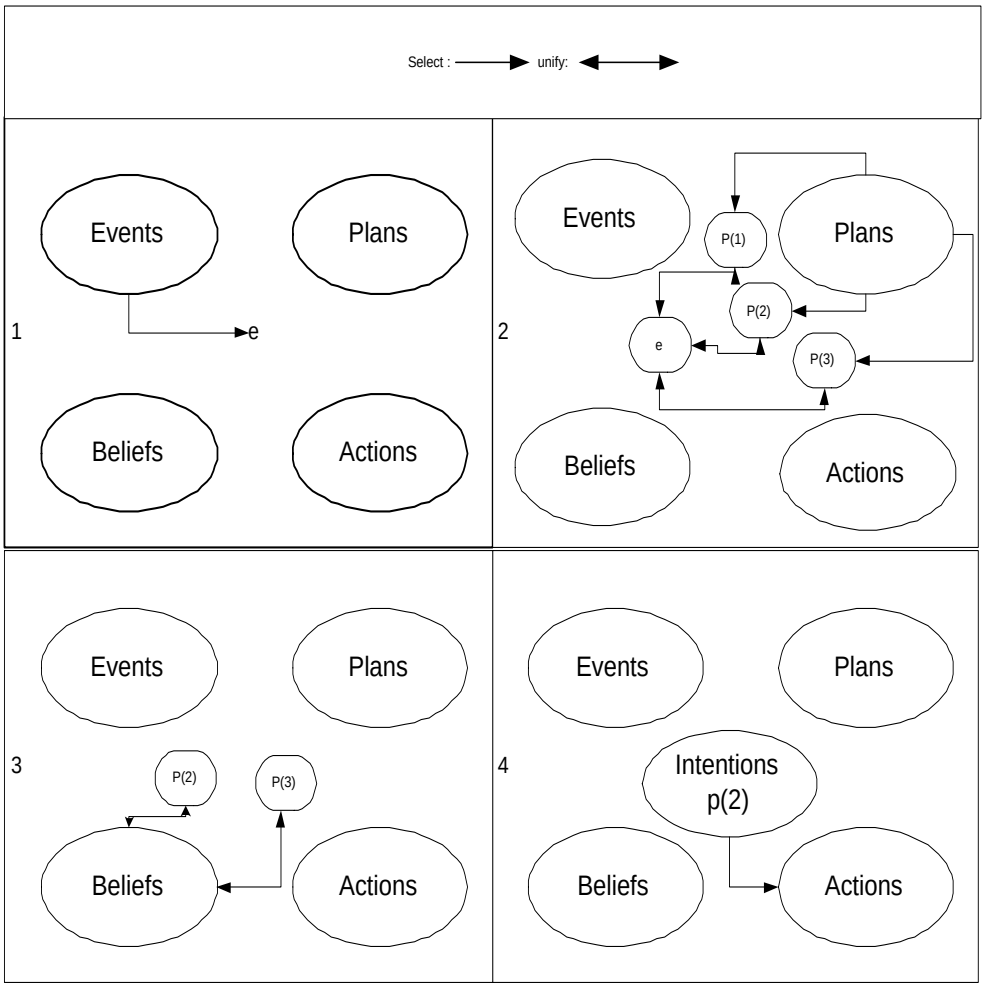


FIG. 6.2 – Fonctionnement d'un agent AgentSpeak(L)

1. Si l'ensemble des événements n'est pas vide alors on sélectionne un event (e).
2. On choisit dans l'ensemble des plans (plan library ou base des plans) les plans (p(1), p(2), p(3)) qui correspondent à l'événement et dont l'événement déclenchant est satisfait. On trouve cette correspondance grâce à l'unification de l'événement avec les plans de la base des plans. Nous avons nommés ces plans (p(1), p(2), p(3)), les plans intéressants.
3. On prend dans l'ensemble des plans (p(1), p(2), p(3)) les plans (p(2),p(3)) dont le contexte (suite de croyances qui sont nécessaires à l'agent pour pouvoir exécuter ce plan) est satisfait, c'est-à-dire unifiable avec les croyances. Nous avons nommés ces plans (p(2),p(3)), les plans appli-

cables.

4. On choisit dans ces plans (p(2),p(3)) un plan (p(2)) que l'on va rajouter à l'ensemble des intentions. Ce plan est alors dit partiellement instancié, ce qui signifie que certaines de ses variables ont pu être liées à des termes. Deux cas sont possibles, soit ce plan sera considéré comme une nouvelle intention car il a été engendré par un **événement externe**, soit ce plan sera considéré comme faisant partie d'une intention déjà existante car il a été engendré par un **événement interne** provenant d'un but de cette intention. Une intention est donc classiquement représentée par une pile de plans.
5. On sélectionne dans l'ensemble des intentions une intention et on considère une par une chaque partie du corps de son top plan. Si on rencontre une **action** alors on la range dans un tampon pour une exécution future. Si on rencontre un **achieve goal** alors on ajoute à l'ensemble des événements un nouvel événement dit interne correspondant et on suspend l'intention jusqu'à ce le but soit satisfait par un autre plan. Si on rencontre un **query goal** alors, si ce but est unifiable avec l'ensemble des croyances, son unificateur le plus général (mgu) est appliqué au reste du plan.
6. Si le top plan de cette intention est terminé alors on considère son plan suivant.
7. Si il n'existe plus de plan suivant alors l'intention a été satisfaite et doit être supprimée de l'ensemble des intentions.

6.3 Comparaison 3APL et AgentSpeak(L)

Premièrement, remarquons que ces deux langages 3APL et AgentSpeak(L) sont basés notamment sur la technologie BDI mais aussi sur un système de règles [18]. Au niveau de 3APL, ces règles sont définies par les practical reasoning rules qui permettent à la fois d'établir des plans et de réviser les buts de l'agent. Au niveau d'AgentSpeak(L), les plans déterminent le savoir faire de l'agent et ont pour fonction unique la planification. Ces plans permettent donc à l'agent de choisir un plan P spécifique, lorsque l'ensemble des croyances de l'agent se limite à B et qu'il doit atteindre le but G. Les plans d' AgentSpeak(L) sont donc moins expressifs que les practical reasoning rules de 3APL car ils servent uniquement à l'établissement de plan et non pas à la révision des buts. De plus, les practical reasoning rules selon leur type peuvent non seulement être utilisées pour les plans mais aussi pour

améliorer la réactivité de l'agent face à certaines situations, l'optimisation de l'exécution de l'agent et la gestion des erreurs.

Deuxièmement, les buts 3APL ont une structure plus complexe que les buts d'AgentSpeak(L). En effet, les buts de ce dernier langage sont soit des achievement goals soit des test goals. Au niveau de 3APL, sont considérés comme buts non seulement les achievement goals et les test goals mais aussi les actions de base et toutes les combinaisons de ses trois types de buts qui s'opèrent grâce aux opérateurs de séquence (;), de choix non-déterministe (+) et de parallélisme(||). *Les concepts d'intention et d'événement définis dans AgentSpeak(L) peuvent donc se ramener au concept de but 3APL* [33]. Remarquons ici, que les notions d'achievement goals, de test goals et d'actions sont quasi identiques pour les deux langages même si la notation est différente. Toutefois, certaines différences existent comme pour la spécification des actions ; car au niveau de 3APL, les impacts qui vont être exercés sur l'état mental de l'agent seront définis au préalable et donc identifiables, tandis que dans les spécifications des actions d'AgentSpeak(L), cette caractéristique est absente.

Troisièmement, en pratique l'exécution des buts d'AgentSpeak(L) se déroule en deux étapes distinctes : processing events et executing intentions. La première étape (Processing events) consiste à sélectionner les plans qui vont être déclenchés par les événements et de les ajouter à l'ensemble des intentions. La deuxième étape (Executing intentions) consiste à exécuter ces intentions et donc les plans correspondant. Ces plans déclenchant à leur tour les actions, test goals ou achievement goals repris dans leur corps. Au contraire, dans 3APL, les règles à appliquer sont choisies durant l'exécution. Il n'existe donc pas d'étape préliminaire pour choisir les meilleures règles avant de les exécuter. L'application des PR-rules consiste principalement soit à exécuter des buts, soit à remplacer un plan par un autre. Ces considérations et la classification des practical reasoning rules permettent à 3APL de fournir une description plus riche du comportement de l'agent même si comme nous l'avons déjà vu cette classification pourrait être améliorée en la rendant plus dynamique.

En conclusion, les moyens mis à disposition par 3APL afin de modifier, de réviser les buts et de gérer les erreurs sont une des qualités majeures de ce langage par rapport à AgentSpeak(L). Ces deux langages utilisent les mêmes principes issus pour la plupart des notions introduites par Shoham à travers son langage Agent-0. La définition des rules 3APL est pourtant assez particulière et plus expressive que celle proposée par AgentSpeak(L). En général, la stratégie d'exécution des agents est similaire pour les deux langages ; dans un premier temps le but est de trouver une règle applicable pour décomposer

un but, ensuite on sélectionne un (sous)but et enfin on exécute ce (sous)but. 3APL et AgentSpeak(L) utilise donc une approche top-down, c'est à dire qu'ils partent d'un but qui va être décomposé en une série de (sous)buts grâce aux règles. Ce processus continuant jusqu'à ce le but de base ai été satisfait ou qu'on ne trouve plus de règles applicables. Ces deux langages utilisent donc le même cycle général d'exécution :

1. Selection d'une règle R à exécuter.
2. Exécution de cette règle R et mise à jour de la base des buts.
3. Sélection d'un but G.
4. Exécution de G.
5. Goto 1.

En outre, il reste à noter que l'article [18] établit à l'aide d'une méthode de preuve formelle que le comportement des agents AgentSpeak(L) est simulable de manière générale avec les agents 3APL.

Chapitre 7

Le langage de programmation Congolog

7.1 Introduction

Après avoir présenté dans le chapitre précédent le langage de programmation AgentSpeak(L), qui est dans une certaine mesure assez proche du langage 3APL, nous allons aborder un langage un peu particulier. Congolog est un langage de programmation basé sur la théorie du situation calculus ou "calcul des situations" et non pas sur la théorie BDI. Cette théorie est habituellement utilisée pour représenter un monde en évolution et fournir un descriptif des aspects dynamiques d'un système. Ce langage est en fait une évolution du langage **Golog** que l'on a voulu rendre entre autre concurrent. Il est axé principalement sur la programmation dans le domaine de la robotique et des agents logiciels. Récemment, une nouvelle évolution appelée **IndiGolog** a vu le jour afin de pallier certains inconvénients de Congolog, mais cette nouvelle approche apporte son lot de limitations.

Dans ce chapitre nous présenterons dans un premier temps la théorie du calcul des situations, ensuite nous développerons les caractéristiques du langage Golog et à partir de là nous pourrons donner une description du langage Congolog illustrée par un exemple concret. Enfin, nous terminerons avec une comparaison entre Congolog et 3APL.

7.2 Calcul des situations

Le calcul des situations utilisé dans ConGolog a été défini par Reiter [34]. C'est avant tout un langage du premier ordre auquel quelques axiomes du second ordre ont été ajoutés. Ce langage est utilisé pour modéliser des environnements en évolution continue. Le principe de base du calcul des situations est de représenter, à tout instant t, l'environnement par la situation dans lequel il se trouve. Le passage d'une situation à l'autre ne pouvant s'accomplir que grâce à l'accomplissement d'une action par un agent.

Si S_0 est la ***situation initiale*** dans laquelle se trouve l'environnement, alors la *situation* S_n est simplement considérée comme la suite d'actions accomplies par les agents pour passer de la situation S_0 à la situation S_n.

Au niveau du calcul des situations, les situations sont représentées par des termes du premier ordre et les actions par des fonctions. De plus, on définit la notion de ***fluent*** comme un terme du premier ordre ou une fonction dont la valeur peut changer d'une situation à l'autre et dont le dernier argument est une situation. C'est pourquoi les fluents sont soit de type fonctionnel $(f(\vec{x}, s) = c)$ soit de type relationnel$(F(\vec{x}, s)$) suivant que l'on ait affaire à une fonction ou à un terme du premier ordre.

Pour faciliter le passage entre les différentes situations, il a été nécessaire d'utiliser une ***fonction successeur*** (action x situation -> situation) définie sur les situations. Cette fonction "do(a,s)" renvoie donc la situation s' successeur de la situation s suite à l'accomplissement de l'action a dans s. De même, un prédicat "Poss(a,s)" (action x situation) est défini de tel sorte qu'on peut affirmer que l'action a peut être exécutée dans la situation s.

Des axiomes fondamentaux de trois types différents doivent être énoncés afin de spécifier un monde dynamique à l'aide du calcul des situations. Ces différents axiomes feront l'objet des sections 7.2.1, 7.2.2 et 7.2.3

7.2.1 Axiomes de précondition

Toutes les actions ne peuvent pas se produire dans toute les situations. Ce type d'axiome est donc utilisé pour décrire toutes les conditions qui sont nécessaires afin d'exécuter une action. Le format des axiomes de précondition est le suivant :

$Poss(A(\vec{x}), s) \Longleftrightarrow \pi_A(\vec{x}, s)$
Où
- $A(\vec{x})$ est une action
- $\vec{x}$ sont les paramètres de l'action
- s est une situation
- $\pi_A(\vec{x}, s)$ est une formule du première ordre qui spécifie les conditions nécessaires et suffisantes pour exécuter l'action. Cette formule est dite uniforme dans la situation s c'est-à-dire qu'elle n'a pour objet que ce qui est vrai dans s.

7.2.2 Axiomes de situation suivante

Ce type d'axiome est utilisé pour spécifier la façon dont le monde évolue. Plutôt que de préciser l'effet qu'a chaque action sur chaque fluent, on pose un axiome par fluent afin de définir récursivement sa valeur selon la situation précédente et l'action amenant à la situation actuelle. En pratique, est associé à chaque fluent un axiome de situation suivante qui spécifie comment la valeur du fluent va réagir par rapport à une action.

Si
- $\vec{x}$ sont les paramètres du fluent
- a est une action
- s est une situation
- γ^+ est une formule du premier ordre uniforme dans s qui spécifie les conditions pour lesquelles le fluent est vrai après que l'action a ait été exécutée.
- γ^- est une formule du premier ordre uniforme dans s qui spécifie les conditions pour lesquelles le fluent est faux après que l'action a ait été exécutée.

Alors
- Le format des axiomes de situation suivante liés aux fluents de type fonctionnel est le suivant :
  $f(\vec{x}, do(a, s)) = c \Longleftrightarrow \gamma^+(\vec{x}, a, s) \vee f(\vec{x}, s) = c \wedge \neg\gamma^-(\vec{x}, a, s)$

- Le format des axiomes de situation suivante liés aux fluents de type relationnel est le suivant :
  $F(\vec{x}, do(a, s)) \Longleftrightarrow \gamma^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma^-(\vec{x}, a, s)$

7.2.3 Axiomes de la situation initiale

Ce type d'axiome est utilisé pour spécifier la valeur des fluents dans la situation initiale. Les initial state axioms sont des formules uniformes dans la situation s.

7.3 Golog

Golog [36] est un langage de programmation orienté-agent basé sur le calcul des situations. En effet, les actions primitives en Golog sont modélisées à travers les préconditions qui sont nécessaires à leur exécution correcte et les impacts qu'elles engendrent sur leur environnement. Les préconditions et les différents impacts étant définis à l'aide des axiomes de précondition et des axiomes de situation suivante. Afin d'avoir un langage complet, expressif et utilisable, il existe différents opérateurs de base entre ces actions.

a	action primitive
c ?	attend la condition c
$(\theta_1; \theta_2)$	séquence d'actions
$(\theta_1 \mid \theta_2)$	choix nondéterministe entre actions
πv.θ	choix nondéterministe des arguments
θ^*	itération nondéterministe
proc $p(\vec{x})\theta$ end	procédure
p($\vec{t}$)	appel de procédure

Le principe général d'exécution d'un programme Golog n'est pas de trouver simplement une séquence d'actions qui permet de satisfaire le but de départ. Mais, en utilisant le programme de haut niveau modélisant le comportement souhaité de l'agent défini par le programmeur, l'interpréteur cherche une séquence d'actions valide qui conduit à un situation finale admissible pour le programme.

7.4 ConGolog

On peut définir le langage de programmation ConGolog [15] comme le langage Golog dans lequel on a introduit la notion d'action exogène et la notion de concurrence (d'où le nom ConGolog). Une action exogène étant une action primitive qui n'a pas été prévue dans le programme mais qui doit être exécutée à cause de l'environnement de l'agent. Ce Golog nouvelle génération entraîne donc l'apparition non seulement d'un nouveau prédicat Exo(a) pour définir les actions exogènes mais aussi de nouveaux opérateurs

liés, entre autre, à la concurrence :

if c then θ_1 else θ_2	Structure if-then-else
while c do θ	Structure While
$(\theta_1 \| \| \theta_2)$	exécution concurrente
$(\theta_1 >> \theta_2)$	priorité au niveau de la concurrence
$\theta^{\|\|}$	itération concurrente
$< c \rightarrow \theta >$	interruption

7.4.1 Exécution concurrente

L'interleaving est la solution qui a été choisie pour modéliser l'exécution concurrente de deux processus. Il n'y a donc jamais qu'une seule action qui s'exécute à la fois. Néanmoins, l'exécution concurrente de processus est rendue possible simplement en bloquant un processus en cours d'exécution pour donner la main à un autre processus. Le processus suspendu peut être débloqué si et seulement si les préconditions de l'action à laquelle il s'est arrêté sont satisfaites ou que le test (c ?) auquel il s'est arrêté est vrai.

7.4.2 Priorité au niveau de la concurrence

Cette expression $(\theta_1 >> \theta_2)$ signifie notamment que θ_1 et θ_2 s'exécutent de manière concurrente mais elle indique aussi à l'interpréteur que θ_2 ne peut être exécuté que si θ_1 est bloqué ou terminé.

7.4.3 Interruptions

Une interruption se modélise de la façon suivante : $< c \rightarrow \theta >$, et signifie que le programme θ sera déclenché si la condition c est satisfaite. Une interruption peut être redéclenchée uniquement quand sa première exécution est terminée. Il est aussi possible de donner un ordre de priorité entre les différentes interruptions.

En pratique, un programmeur voulant modéliser un système multi-agents doit donc fournir pour chaque action les axiomes de précondition, pour chaque fluent les axiomes d'états successeur, une spécification de la situation initiale et un programme ConGolog spécifiant le comportement des agents du système.

7.4.4 Communication entre agents

Il faut noter que des outils de communications propres à ConGolog ont été définis via un fluent lié à la gestion d'une file de messages en attente pour chaque agent et via des actions primitives opérant sur ces files de messages [35]. Chaque agent possède un fluent fonctionnel FileMess(agt,s) qui représente une file de messages. Trois actions primitives opèrent sur cette file de messages :

1. EnvoieMess(destinataire, mess) : envoyer le message mess à l'agent destinataire, ce qui a pour effet d'ajouter la paire <expéditeur, mess> à la file de cet agent ;
2. InspecteMess : percevoir quelle paire <expéditeur, mess> est à la tête de sa file ;
3. EnlèveMess : enlever la paire <expéditeur, mess> qui est à la tête de sa file.

Seul un axiome de précondition est nécessaire car d'une part, l'action EnlèveMess nécessite que la file de message de l'agent soit non vide et d'autre part, les deux autres actions sont toujours accessibles.

Poss(agt,EnlèveMess,s) $\equiv \neg Vide$(FileMess(agt,s)) ;

L'axiome de situation suivante pour le fluent FileMess est donc le suivant :

FileMess (agt′,do(agt,act,s))=q ≡
$\exists m$ (act=EnvoieMess(agt′,m)∧ q = Concatène(FileMessage(agt′,s),[<agt,m>]))
∨ act = EnlèveMess ∧ agt = agt′ ∧ q = Reste(FileMess(agt′,s))
∨ q = FileMess(agt′,s) ∧
$\forall m$ (act ≠ EnvoieMess(agt′,m)∧(act ≠ EnlèveMess ∨ agt ≠ agt′)

Ce mode de communication a été développé pour palier aux problèmes liés au langage de communication KQML, la question est de savoir si l'apparition d'autres langages de communication tel que FIPA n'aurait pas pu convenir ? Car il est sans aucun doute plus intéressant d'avoir des agents concurrents dotés de moyens de communication standardisés tel que FIPA plutôt que des agents limités car obligés d'utiliser les moyens de communication fournis par ConGolog. L'utilisation de FIPA permettrait d'améliorer l'interopérabilité entre les différents agents.

7.5 Exemple : Contrôleur d'ascenseur

Cet exemple consiste simplement à gérer le contrôleur d'un ascenseur sans se préoccuper ni de la gestion de l'ouverture ou de la fermeture des portes ni des passagers. Nous allons donc ici vous présenter une solution [15] qui illustre parfaitement les différents concepts et notions vus plus haut.

En supposant que le symbole e représente un ascenseur et que le numéro des différents étages est compris entre 1 et 6, la solution se structure comme suit :

Actions Primitives

goDown(e)	l'ascenseur e descend d'un étage
goUp(e)	l'ascenseur e monte d'un étage
buttonReset(n)	éteint le bouton de l'étage n
toggleFan(e)	change l'état du ventilateur de e
ringAlarm	sonne l'alarme liée à la fumée

Actions Exogènes

reqElevator(n)	appuyer sur le bouton de l'étage n
changeTemp(e)	la température de l'ascenseur change
detectSmoke	le détecteur de fumée s'enclenche
resetAlarm	éteint l'alarme liée à la fumée

Primitive Fluents

floor(e,s)=n	l'ascenseur e est à l'étage n
temp(e,s)=t	la température de l'ascenseur e est de t
FanOn(e,s)	le ventilateur de l'ascenseur e est activé
Button(n,s)	le button de l'étage n est activé
Smoke(s)	de la fumée a été détectée

Defined Fluents

TooHot(e,s)= temp(e,s) $>$ 1	la température de e est trop élevée
TooCold(e,s)= temp(e,s)$<$ -1	la température de e est trop basse

Axiomes de la situation initiale

floor(e,S_0)=1
¬FanOn(S_0)
temp(e,S_0)=0
ButtonOn(3,S_0), ButtonOn(6,S_0)

Axiomes des actions exogènes

$\forall a.Exo(a) \equiv$ a = detectSmoke $\vee$ a= resetAlarm $\vee$
$\exists e$.a=changeTemp(e) $\vee$ $\exists n$.a = reqElevator(n)

Axiomes de précondition

Poss(goDown(e),s) $\equiv$ floor(e,s) $\neq 1$
Poss(goUp(e),s) $\equiv$ floor(e,s) $\neq 6$
Poss(buttonReset(n),s) $\equiv$ True
Poss(toggleFan(e),s) $\equiv$ True
Poss(ringAlarm) $\equiv$ True
Poss(reqElevator(n),s) $\equiv (1 \leq n \leq 6) \wedge \neg ButtonOn(n, s)$
Poss(changeTemp,s) $\equiv$ True
Poss(detectSmoke,s) $\equiv \neg Smoke(s)$
Poss(resetAlarm,s) $\equiv$ Smoke(s)

Axiomes de situation suivante

floor(e,do(a,s))=n ≡
(a = goDown(e) ∧n = floor(e,s)-1) ∨
(a = goUp(e) ∧ n = floor(e,s)+1) ∨
(n = floor(e,s) ∧ a ≠ goDown(e) ∧ a ≠ goUp(e))
temp(e,do(a,s))=t ≡
(a = changeTemp(e) ∧ FanOn(e,s) ∧ t = temp(e,s)-1) ∨
(a = changeTemp(e) ∧ $\neg FanOn(e, s)$ ∧ t = temp(e,s)+1) ∨
(t=temp(e,s) ∧ $a \neq$ changeTemp(e))
FanOn(e,do(a,s)) ≡
(a = toggleFan(e) ∧ ¬FanOn(e,s)) ∨
(FanOn(e,s) ∧ a ≠ toggleFan(e))
Button(n,do(a,s)) ≡
(a = reqElevator(n) ∨
(ButtonOn(n,s) ∧ a ≠ buttonReset(n))
Smoke(do(a,s)) ≡
a = detectSmoke ∨
(Smoke(s)∧ a ≠ resetAlarm)

Programme Congolog

A partir de ces axiomes on peut désormais implémenter un simple contrôleur d'ascenseur grâce au langage Congolog.

While ∃n.ButtonOn(n) **do** π n.BestButton(n)?; serveFloor(e,n);
While floor(e) ≠ 1 **do** goDown(e)

Il est à noter que si aucun bouton n'est activé au départ, la première boucle du programme se terminera et l'ascenseur descendra au premier étage et s'y arrêtera même si d'autres boutons sont activés pendant sa descente. Les interruptions comme nous le verrons dans la suite peuvent facilement répondre à ce problème.

Le fluent BestButton pourrait être défini comme le ButtonOn qui a été activé depuis le plus longtemps, mais dans cette solution les auteurs ont considéré simplement qu'il n'y avait aucune priorité entre les différents ButtonOn.

La procédure serveFloor(e,n) détermine les actions qu'il est nécessaire d'accomplir par l'ascenseur e pour desservir l'étage n.

proc serveFloor(e,n)
While floor(e) < n **do** goUp(e) ;
While floor(e) > n **do** goDown(e) ;
buttonReset(n)
end

Pour affiner le comportement du contrôleur, il est très pratique de se servir des interruptions mises à disposition par Congolog. Dans cet exemple on peut notamment les utiliser pour gérer quatre cas différents en leur donnant aussi un ordre de priorité :

1. Gérer la mise route ou l'arrêt des ventilateurs de l'ascenseur suivant le niveau de sa température. Ce cas ayant nécessairement la priorité la plus élevée.
   <TooHot(e) ∧ ¬FanOn(e)→ toogleFan(e)>
   <TooCold(e) ∧ FanOn(e)→ toogleFan(e)>

2. Donner une priorité plus importante pour pouvoir évacuer en cas d'urgence. On utilise donc un nouveau fluent EButtonOn(n) et une nouvelle procédure serveEFloor, le E signifiant Emergency.

3. Gérer le déclenchement de l'alarme liée à la fumée.
   <Smoke → ringAlarm>
4. Gérer le cas de départ où aucun bouton n'est activé. En effet, si aucun bouton n'est activé au départ, la première boucle du programme se terminera et l'ascenseur descendra au premier étage et s'y arrêtera même si d'autres boutons sont activés pendant sa descente. Les interruptions peuvent facilement répondre à ce problème.
   <∃n.ButtonOn(n) → πn.{BestButton(n) ? ;serveFloor(e,n)}>
   <floor(e)≠ 1 → goDown(e)>

Les priorités au niveau de la concurrence des différentes interruptions sont donc logiquement définies comme suit :

(<TooHot(e) ∧ ¬FanOn(e)→ toogleFan(e)>||
<TooCold(e) ∧ FanOn(e)→ toogleFan(e)>)»
<∃n.EButtonOn(n) → πn.{EButtonOn(n) ? ;serveEFloor(e,n)}> »
<Smoke → ringAlarm>»
<∃n.ButtonOn(n) → πn.{BestButton(n) ? ;serveFloor(e,n)}>»
<floor(e)≠ 1 → goDown(e)>

7.6 Comparaison 3APL et Congolog

L'intérêt de ce chapitre est d'exposer un langage de programmation orienté-agent assez différent de 3APL au premier abord. En effet, les langages Congolog et 3APL sont basés d'un coté sur la théorie du calcul des situations et de l'autre sur la théorie BDI.

La différence la plus importante entre ces deux langages se situe au niveau du modèle d'exécution. En effet, l'interpréteur ConGolog détermine grâce au programme ConGolog de haut niveau une séquence d'actions qui définit le comportement de l'agent de manière statique. Tandis que l'interpréteur transforme les buts de l'agent dynamiquement grâce aux PR-rules afin de satisfaire le but de départ.

D'un autre point de vue [33], il est à souligner que le programme ConGolog de haut niveau reste tout de même assez analogue avec la notion de but 3APL. De plus, les différentes situations définies à l'aide du calcul des situations peuvent être vues comme les croyances de l'agent. Par rapport à ce que l'on pourrait penser à première vue, le langage Congolog n'est pas si éloigné du langage 3APL.

D'autres différences existent entre ces deux langages, nous ne pouvons pas toutes les énoncer, mais voici un bref aperçu des plus importantes.

Comme nous l'avons déjà vu, les tests 3APL définis par les Test Goals peuvent donner trois types de résultats différents : vrai, faux ou indéterminé. Ce dernier résultat est lié à l'absence d'informations suffisantes pour que le test demandé à la base des croyances puisse fournir une réponse vraie ou fausse. Au niveau de Congolog, cette particularité n'existe pas car les tests ne peuvent être que vrais ou faux suivant la situation courante.

L'opérateur de choix nondéterministe πx est spécifique au langage Congolog. Cette opérateur liant explicitement les variables du programme. Tandis qu'au niveau de 3APL ce mécanisme est implicitement réalisé grâce au Test Goal.

7.7 Conclusions

Congolog est un langage de programmation permettant de spécifier précisément un système et ses interactions avec l'environnement. Une caractéristique très intéressante de ce langage est sa capacité à modéliser et raisonner sur de l'information incomplète. Par contre, l'exécution de l'interpréteur ne peut pas se faire en temps réel, en effet, il doit, avant de pouvoir exécuter la moindre action, trouver une séquence d'actions qui permettra de passer de la situation initiale à la situation finale espérée. De plus, dans le langage ConGolog l'agent n'a aucun moyen d'obtenir en temps réel des renseignements sur son environnement grâce à des senseurs, les actions exogènes étant définies au préalable. Ce manque de capacité d'adaptation aux changements pouvant se produire dans l'environnement est une limitation importante.

Ce problème étant, en fait, une des raisons d'être du développement du langage IndiGolog [24], qui permet justement d'utiliser l'interpréteur en temps réel. Cette nouvelle évolution est passée à une exécution dite incrémentale de l'interpréteur, c'est-à-dire que les actions sont exécutées avant même d'avoir trouver une séquence complète d'actions. Le système exécute simplement les actions déterminées par le programme et dont toutes les préconditions sont satisfaites. En outre, pour optimiser les séquences d'actions de l'agent, la planification en ligne est nécessaire. Indigolog introduit aussi la notion des sensing actions afin de permettre à l'agent d'avoir un lien direct avec son environnement. Toutefois, cette exécution incrémentale a un inconvénient majeur lié au fait qu'il est impossible pour l'interpréteur de retrouver la situation précédente dans lequel le système se trouvait. Si un plan n'arrive pas à satisfaire un but, le système sera dans une impasse car il sera impossible de retrouver le but de départ et donc de sélectionner un nouveau plan. Le backtracking utilisé au niveau de Congolog n'est donc plus possible dans le cadre d'IndiGolog.

Conclusions

Pour conclure cette troisième partie, on se rend compte que depuis le début de la recherche orientée-agent, différents langages de programmation ont été proposés afin d'implémenter des versions exécutables de systèmes orientés-agent. Les premiers langages de programmation à avoir été utilisés sont C, C++, Prolog. Au jour d'aujourd'hui, JAVA est certainement le langage le plus populaire parmi la communauté des programmeurs d'agents. Toutefois, au départ, JAVA n'a rien d'un langage destiné à la conception de système orienté-agent. De nombreuses nouvelles fonctionnalités ont donc du être développées par les programmeurs eux-même. De plus, en utilisant ce langage orienté-objet, les programmeurs ne peuvent concevoir leurs agents directement car ils doivent penser avant tout en terme d'objet. Actuellement, cette programmation de type *"agent-like programming"* est la plus en vogue, ce qui ne paraît pas très intéressant d'un point de vue conceptuel. Les raisons pour lesquels JAVA détient un quasi monopole sur la programmation orientée-agent sont de plusieurs ordres ; premièrement, le phénomène de mode associé à ce langage n'est pas à négliger, deuxièmement les librairies JAVA fournissent aux programmeurs un outil puissant, troisièmement JAVA est caractérisé par la facilité avec laquelle on peut lui rajouter de nouvelles fonctionnalités et enfin, il est difficile pour les programmeurs de changer leurs habitudes en passant à un nouveau langage qui est malgré tout toujours en cours de développement.

En effet, on assiste à l'émergence de nouveaux types de langages tel que AgentSpeak(L), Congolog et 3APL, mais qui sont encore peu usités. Ces langages ont pour objectif premier et essentiel de permettre aux programmeurs de concevoir directement leur système d'agents en terme d'agent et non d'objet. Ces différents langages ont chacun leurs propres avantages.

Ainsi ConGolog a l'avantage de pouvoir, non seulement, gérer de l'information incomplète, mais aussi d'affiner la modélisation du comportement de l'agent grâce aux concepts d'interruption et de prioritization.

D'un autre côté, 3APL peut s'exécuter en temps réel et surtout permet au

programmeur d'avoir une approche plus intuitive pour spécifier le comportement de ses agents grâce notamment aux PR-rules. De plus, 3APL fournit une solution simple pour modéliser les interactions qu'un agent doit avoir avec son environnement via ses senseurs et ses effecteurs.

Outre les différents inconvénients propres à chacun de ces langages, un problème majeur persiste puisqu'au niveau de la communication entre agents, ils utilisent des moyens de communication qui leur sont propres. Ce manque de standardisation est un frein important à leur émancipation. L'utilisation d'un standard tel que FIPA est sans doute une bonne alternative pour pallier ce manque d'interopérabilité.

Quatrième partie

Etude de cas

Chapitre 8

Développement d'un programme 3APL grâce à la méthodologie AAII

Dans ce chapitre, nous allons tenter de fournir un exemple de développement de système multi-agents, depuis la description informelle du système jusqu'à l'implémentation. Pour ce faire, nous utiliserons la méthodologie AAII présentée dans le chapitre 2, qui aboutira à un modèle conceptuel du système. Nous traduirons ensuite ce modèle dans le langage 3APL. Mais avant toute chose, nous allons fournir la description informelle du système à implémenter, ainsi que certaines hypothèses simplificatrices.

8.1 Description informelle

L'application est un système de négociation commerciale simplifié, dont le but est de permettre à un acheteur et un vendeur de s'accorder sur le prix d'un produit. L'acheteur maintient une liste de produits qu'il veut acheter, ainsi que pour chacun de ces produits le prix maximal qu'il consentirait à payer. Le vendeur quant à lui maintient la liste des produits qu'il propose, ainsi que pour chacun de ces produits le prix minimal auquel il consentirait à le vendre. Afin de convenir d'un prix, acheteur et vendeur se font mutuellement une série de propositions et de contre-propositions, jusqu'au moment où l'un des acteurs accepte la proposition de l'autre. L'acceptation d'un proposition se fait, pour l'acheteur, si le prix proposé par le vendeur est inférieur à son prix maximal. L'acceptation d'une proposition se fait par le vendeur si le prix proposé par l'acheteur est supérieur à son prix minimal. De plus, vendeur et acheteur peuvent faire une proposition "à prendre ou à laisser", lorsqu'ils

sont amenés à proposer respectivement leur prix minimal ou maximal. Une telle proposition ne peut être qu'acceptée ou rejetée.

8.2 Hypothèses simplificatrices

- le système ne gère que la négociation du prix d'un produit. Tous les aspects relatifs à la transaction et à la livraison du produit ne sont pas gérés. Ainsi, un produit est considéré comme acheté par l'acheteur si celui-ci est parvenu à négocier un prix avec le vendeur. De même, un produit est considéré comme vendu par le vendeur, si celui-ci est parvenu à négocier un prix avec l'acheteur
- le système ne comporte à l'exécution qu'un acheteur et un vendeur.
- acheteur et vendeur essayent d'obtenir respectivement le prix le plus bas et le prix plus haut possible. Toutefois, à partir du moment où une proposition qui leur convient leur est faite, ils l'acceptent, et n'essaient plus d'en tirer un prix plus avantageux.
- l'initiateur de la négociation est toujours l'acheteur

8.3 Application de la méthodologie AAII

Les rôles du systèmes sont les rôles ACHETEUR et VENDEUR.

La responsabilité du rôle ACHETEUR est d'acheter un certain nombre de produits à un prix inférieur à un prix donné. Les services qui mettent en oeuvre cette responsabilité sont les suivants :

- faire une proposition de prix pour un produit au vendeur ($proposition\_achat(prod, prix)$). Nous pouvons donc en dériver le but d'obtenir le prix $prix$ pour le produit $prod$ ($achieve(prixNegocie(prod, prix))$).
- accepter une proposition faite par le vendeur ($accept\_achat(prod, prix)$). Une fois la proposition acceptée, le but ultime de l'acheteur est rempli, en l'occurence celui de posséder le produit ($achieve(possede(prod))$)
- faire une proposition à prendre ou à laisser ($dernier\_prix\_achat(prod, prix)$). Le but dérivé de cette interaction est le même que celui défini dans le premier point.
- rejeter une proposition à prendre ou à laisser du vendeur ($reject\_achat(prod, prix)$). Une fois la proposition rejetée, l'acheteur aura pour but de considérer le produit trop cher ($achieve(tropCher(prod))$)

La responsabilité du rôle VENDEUR est de vendre un certain nombre de produits à un prix supérieur à leur prix minimum. Les services qui mettent en oeuvre cette responsabilité sont les suivants :

- faire une proposition de prix pour un produit à l'acheteur ($proposition\_vente(prod, prix)$). Nous pouvons donc en dériver le but d'obtenir le prix $prix$ pour le produit $prod$ ($achieve(prixNegocie(prod, prix))$)
- accepter une proposition faite par l'acheteur ($accept\_vente(prod, prix)$). Une fois la proposition acceptée, le but ultime du vendeur est rempli, en l'occurence celui d'avoir vendu le produit ($achieve(vendu(prod))$)
- faire une proposition à prendre ou à laisser ($dernier\_prix\_vente(prod, prix)$). Le but dérivé de cette interaction est le même que celui défini dans le premier point.
- rejeter une proposition à prendre ou à laisser de l'acheteur ($reject\_vente(prod, prix)$). Une fois la proposition rejetée, le vendeur aura pour but de considérer le produit comme invendable ($achieve(nonVendable(prod))$)
- avertir l'acheteur que le produit qu'il a demandé n'est pas disponible ($non_dispo(prod)$). Aucun but n'est associé à ce service pour le vendeur (il se comporte ici de manière réactive). Toutefois, ce service permet de dégager un nouveau but pour l'acheteur, en l'occurence celui de considérer le produit comme non-disponible ($achieve(nonDispo(prod))$)

Notons que les buts dégagés ici permettrons de définir les croyances ultérieurement. En effet les termes se trouvant entre les parenthèses de $achieve()$ sont des croyances de l'agent (comme nous l'avons défini dans le chapitre 2).

Maintenant que nous avons dégagé les différents rôles et services, nous pouvons élaborer les modèles externes, c'est-à-dire le *modèle d'agents* et le *modèle d'interactions*.

8.3.1 Modèles externes

Le modèle d'agents

Etant donné que le modèle d'agent est assez simple, nous allons spécifier le *modèle de classes d'agent* et le *modèle d'instances d'agent* dans le même schéma. De plus nous ferons déjà référence aux modèles de croyances, de buts (ou goals) et de plans définis plus loin.

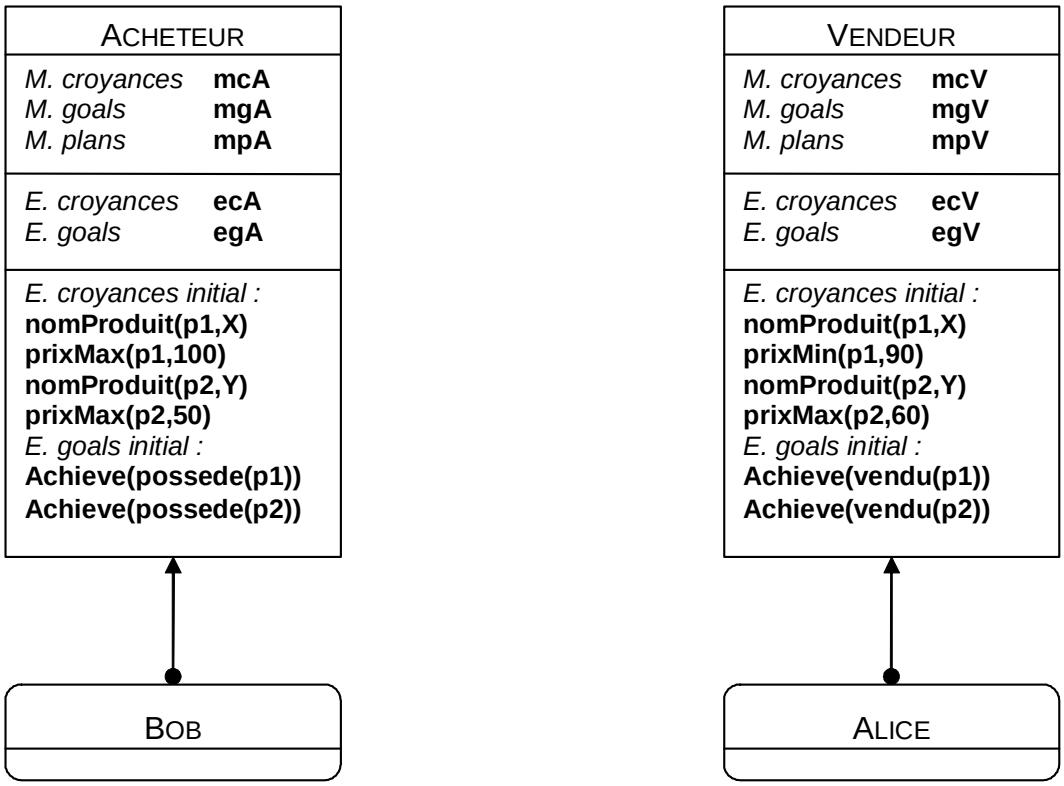


FIG. 8.1 – Modèle d'agents.

Le modèle d'interactions

Comme nous l'avons précisé dans le chapitre 2, la méthodologie AAII ne fournit pas de norme sur le modèle d'interactions. Nous avons choisi de modéliser les interactions grâce à MSC et HMSC, tels qu'ils sont utilisés par la méthodologie MAS-CommonKADS.

Ainsi, nous avons un diagramme défini grâce à HMSC (figure 8.2), qui spécifie le déroulement du protocole. Dans ces diagrammes sont référencées des interactions simples définies grâce à MSC (figures 8.3 et 8.4).

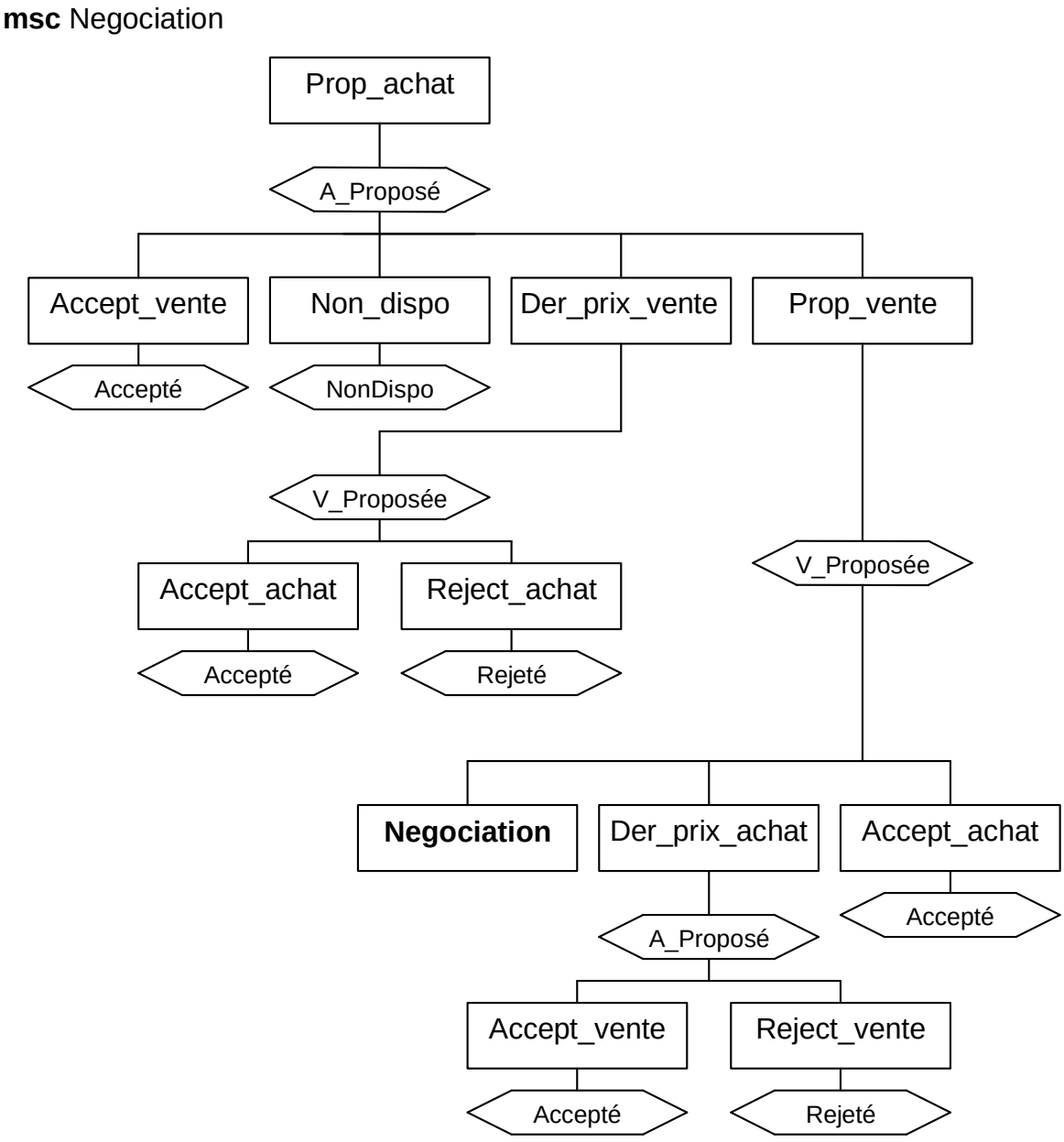


FIG. 8.2 – Diagramme HMSC Negociation.

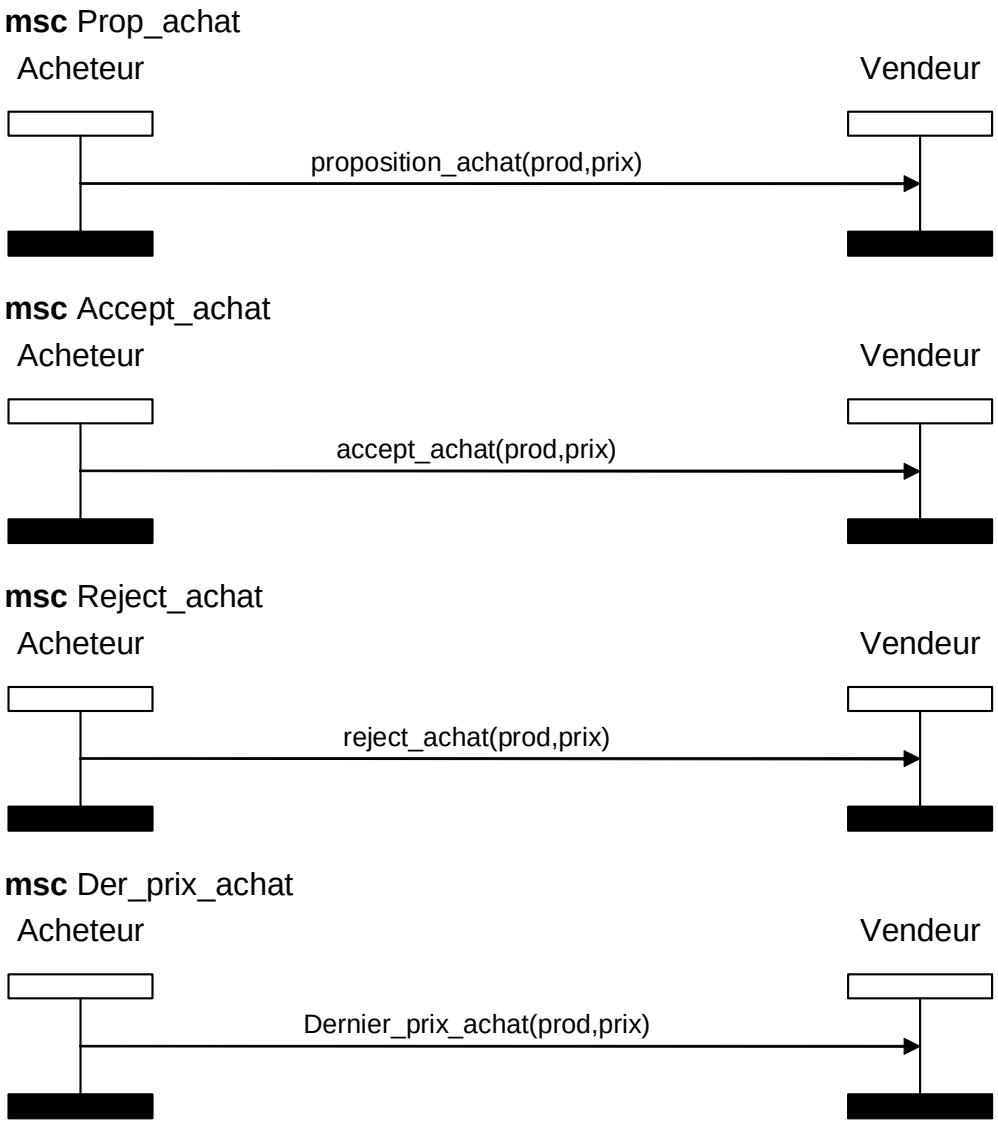


FIG. 8.3 – Messages Acheteur -> Vendeur en MSC.

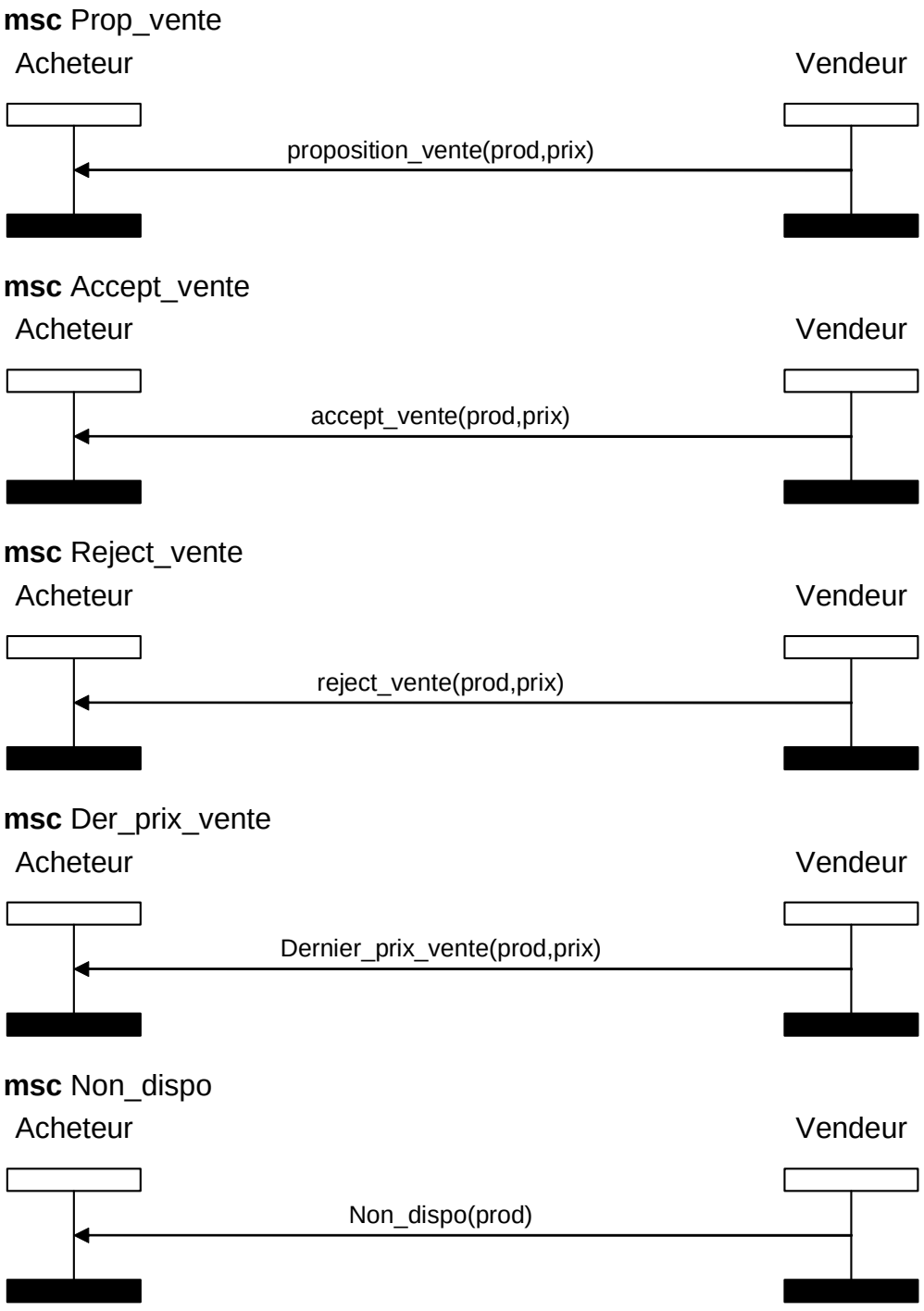


FIG. 8.4 – Messages Vendeur -> Acheteur en MSC.

8.3.2 Modèles internes

Modèles de buts

Le modèle de but de l'acheteur (**mgA**) est le suivant :

```
achieve(possede(Produit))
achieve(prixNegocie(Produit,Prix))
achieve(tropCher(Produit))
achieve(nonDispo(Produit))
```

Le modèle de but du vendeur (**mgV**) est le suivant :

```
achieve(vendu(Produit))
achieve(prixNegocie(Produit,Prix))
achieve(nonVendable(Produit))
```

Ces deux modèles spécifient donc les différents types de buts que peuvent adopter respectivement l'acheteur et le vendeur. Nous ne spécifierons pas les états de buts, ceux-ci ne représentent que les buts initiaux possibles pour les agents qui instancient le type d'agent ACHETEUR et VENDEUR.

Modèles de plans

Le modèle de plans de la classe d'agent ACHETEUR (**mpA**) possède quatre plans, un pour chaque type de buts à accomplir. Ces plans sont présentés aux figures 8.5, 8.6 et 8.7. Notons que les messages envoyés au vendeur et reçus de celui-ci sont en italique.

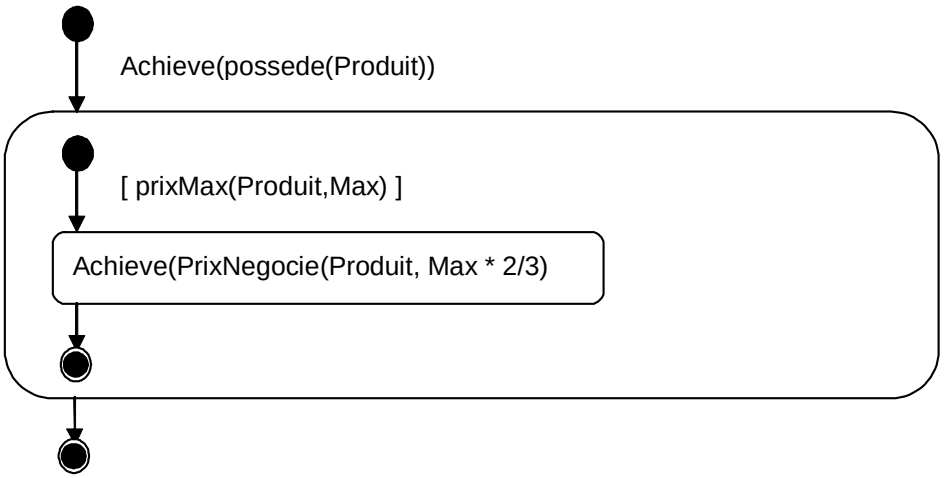


FIG. 8.5 – Plan du type d'agent ACHETEUR.

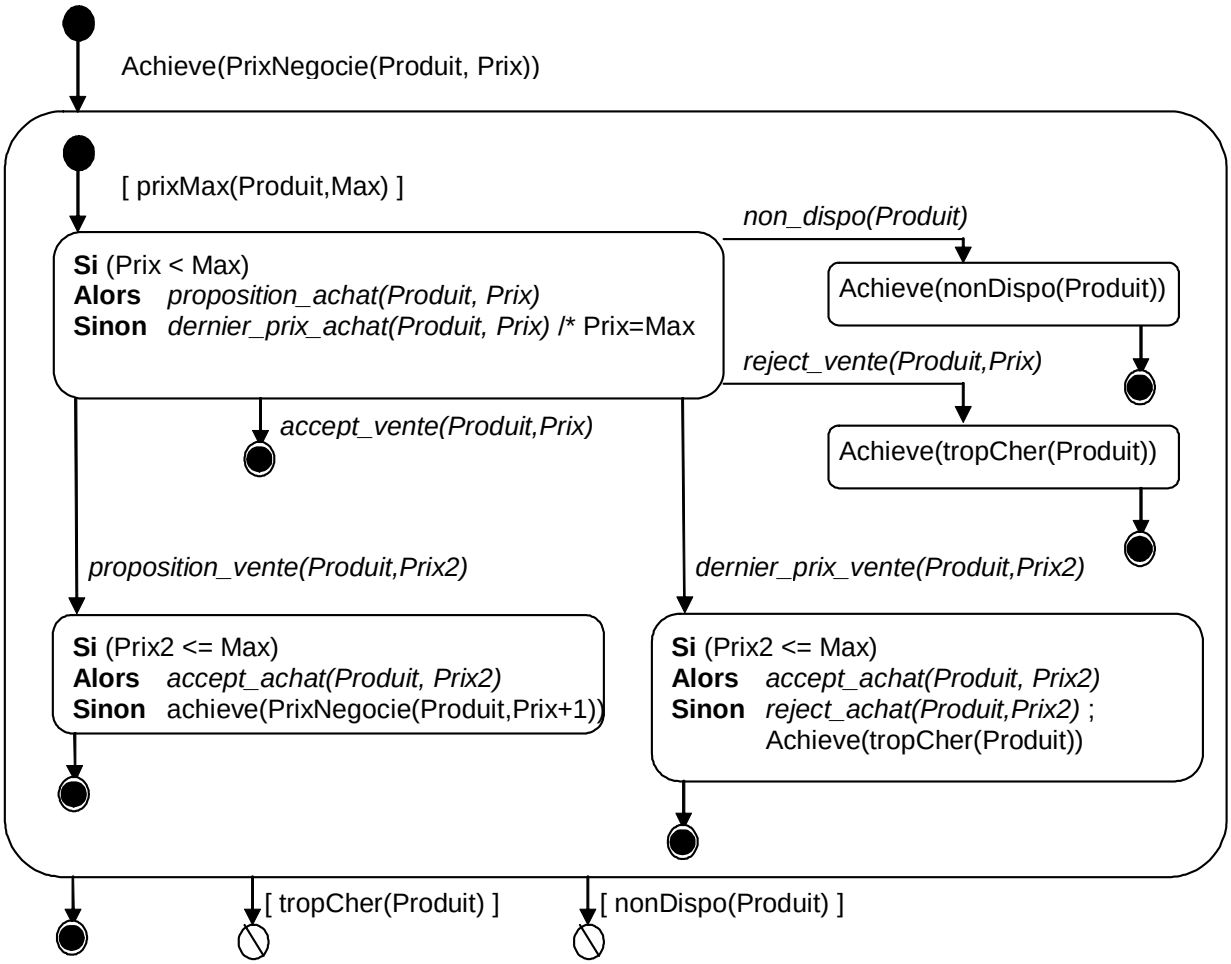


FIG. 8.6 – Plan du type d'agent ACHETEUR.

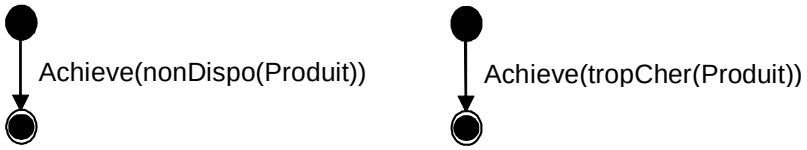


FIG. 8.7 – Plans du type d'agent ACHETEUR.

Les plans présentés à la figure 8.7 n'ont pour but que de rendre une croyance vraie (en l'occurence, $tropCher(Produit)$ ou $nonDispo(Produit)$). Ainsi, le seul fait que le plan réussisse rend le prédicat à l'intérieur des parenthèses de $achieve()$ vrai.

Le modèle de plans de la classe d'agent VENDEUR (**mpV**) possède également quatre plans. Trois d'entre-eux servent à accomplir ses trois types de buts, et le dernier est un plan "réactif", qui indique à l'acheteur qu'un produit n'est pas disponible. Ces plans sont présentés aux figures 8.8, 8.9, 8.10 et 8.11.

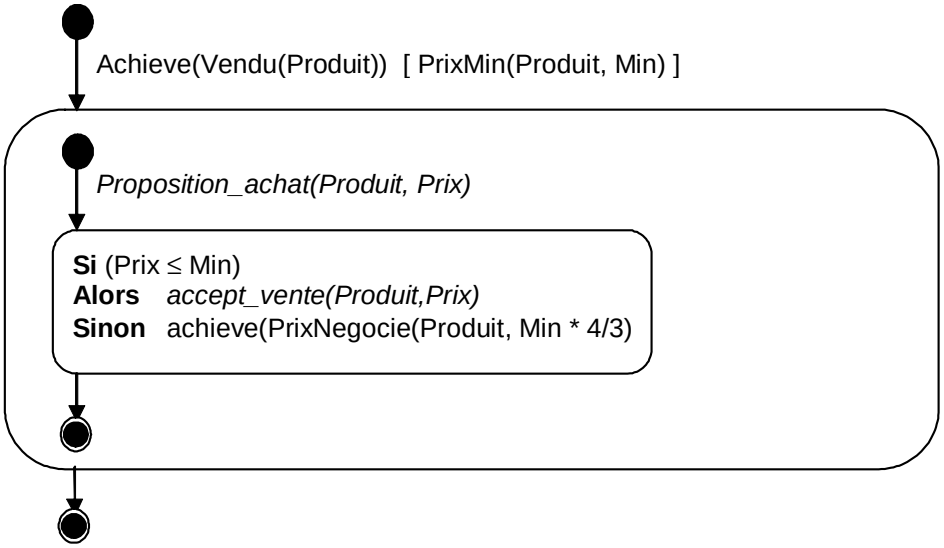


FIG. 8.8 – Plan du type d'agent VENDEUR.

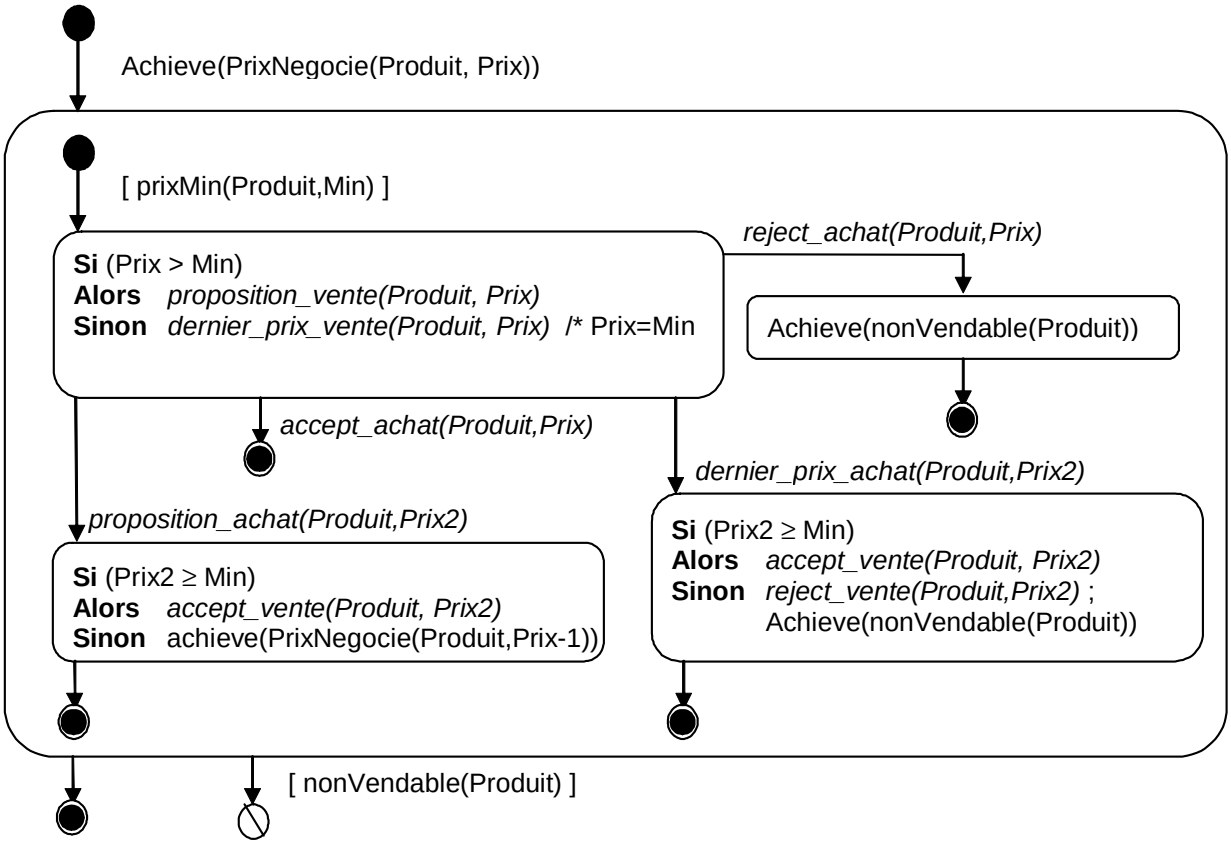


FIG. 8.9 – Plan du type d'agent VENDEUR.

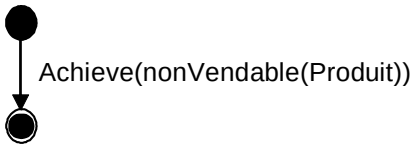


FIG. 8.10 – Plan du type d'agent VENDEUR.

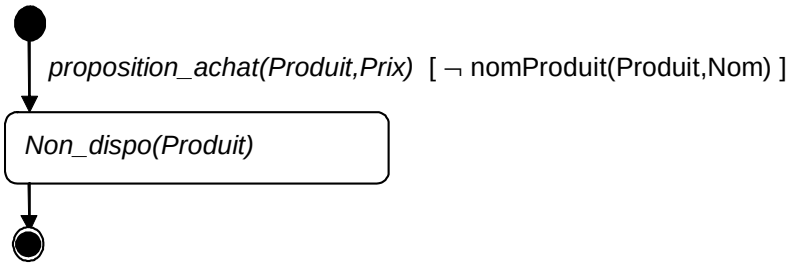


FIG. 8.11 – Plan du type d'agent VENDEUR.

Modèles de croyances

Les modèles de croyances peuvent être dérivés des modèles de goals et de plans définis précédemment. Ainsi, la classe de croyance PRODUIT de l'agent ACHETEUR aura comme attribut le booléen *possede* car il a le goal $achieve(possede(produit))$ (pour rappel, $achieve(X)$ signifie que l'agent a pour but de rendre X vrai), et elle aura également l'attribut $prixMax$, qui est nécessaire au déroulement de ses plans.

Le modèle de croyance de l'acheteur (**mcA**) se trouve à la figure 8.12, et le modèle de croyances du vendeur (**mcV**) se trouve à la figure 8.13. Notons que nous ne dériverons que les prédicats des classes de croyances, car les fonctions d'accès nous sont inutiles. Nous n'allons également pas définir ici les états de croyances de l'acheteur et du vendeur (**ecA** et **ecV**), car ils ne représentent seulement que leurs états de croyances initiaux possibles.

PRODUIT		
Name	nomProduit	nomProduit(Produit,Name)
Boolean	possede	possede(Produit)
Integer	prixMax	prixMax(Produit,Integer)
Integer	prixNegocie {opt}	prixNegocie(Produit,Integer)
Boolean	tropCher	tropCher(Produit)
Boolean	nonDispo	nonDispo(Produit)

FIG. 8.12 – Modèle de croyances de l'acheteur.

PRODUIT		
Name	nomProduit	nomProduit(Produit,Name)
Boolean	vendu	vendu (Produit)
Integer	prixMin	prixMin(Produit,Integer)
Integer	prixNegocie {opt}	prixNegocie(Produit,Integer)
Boolean	nonVendable	nonVendable(Produit)

FIG. 8.13 – Modèle de croyances du vendeur.

8.4 Dérivation du programme 3APL

Au terme de l'application de la méthodologie AAII, nous avons obtenu plusieurs modèles qui puissent être traduit d'une manière relativement aisée en 3APL.

En effet, à chaque instance d'agent du modèle d'agent correspondra évidemment un agent 3APL. L'état de croyance initial de chaque agent constituera leur $BeliefBase$, et leur état de goal initial constiuera leur $GoalBase$. A partir leur modèle de plans, nous pourrons dériver leurs différentes PR-rules. En effet, une PR-rule 3APL a une structure similaire à un plan tel que défini dans la méthodologie AAII : une PR-rule s'active pour un certain goal, moyennant éventuellement une condition sur les croyances de l'agent, et effectue certaines actions.

Néanmoins, certaines modifications sont nécessaires. Premièrement, une transition étiquetée par un message provenant d'un autre agent dans un plan AAII sera transformée par une condition sur les croyances en 3APL. En effet, lorsqu'un agent envoie un message à un autre agent, il ajoute en fait un prédicat à la base des croyances de ce dernier (grâce à l'action $tell()$). Ainsi, des nouveaux prédicats sont nécessaires, qui sont en fait les messages définis dans le modèle d'interactions. Il faudra également retirer ces "messages" de la base de croyance une fois que l'agent les a traités. Deuxièmement, en 3APL, les goals (ou plus précisément les "achievement goals") ne consistent pas forcement à rendre une croyance vraie. En effet, dans la méthodologie AAII, le goal $achieve(vendu(Produit))$ par exemple, rend le prédicat $vendu(Produit)$ vrai si un plan mettant en oeuvre ce goal réussi (ou si le prédicat était déjà vrai). En 3APL par contre, les goals n'ont pas de lien direct avec les croyances, mais toutefois, ils ont la même syntaxe. Nous allons donc garder la même formulation pour les goals que nous avons dégagés précedemment (en retirant toutefois l'opérateur $achieve()$), mais nous devrons explicitement ajouter les prédicats relatifs à ces goals dans la base croyance lorsque ces derniers sont accomplis avec succès. Pour finir, il sera parfois nécessaire de scinder un plan AAII en plusieurs PR-rules 3APL. Ainsi des sous-goals "de service" seront nécessaires, comme nous le verrons par la suite.

Nous pouvons maintenant définir les programmes 3APL de chacun des deux agents.

8.4.1 Agent Acheteur (Bob)

```
CAPABILITIES:

   { }
       assert(X)
   { X }.

   { X }
       remove(X)
   { }.

BELIEFBASE:

   nomProduit(p1,toto).
   prixMax(p1,100).
   nomProduit(p2,tutu).
   prixMax(p2,50).

GOALBASE:

   Possede(p1).
   Possede(p2).

RULEBASE:

   Possede(Produit) <- prixMax(Produit,Max) |
       PrixNegocie(Produit, Max * 2/3).

   PrixNegocie(Produit,Prix) <- prixMax(Produit,Max) |
       IF [Prix < Max]
       THEN tell(Alice,propose_achat(Produit,Prix))
       ELSE
           BEGIN
               tell(Alice,dernier_prix_achat(Produit,Prix));
               SubPrixNegocie(Produit,Prix,Max)
           END.
```

```
SubPrixNegocie(Produit,Prix,Max)
    <- proposition_vente(Produit,Prix2)|
       BEGIN
            remove(proposition_vente(Produit,Prix2));
            IF [Prix2 <= Max]
            THEN
                BEGIN
                    tell(Alice,accept_achat(Produit,Prix2));
                    assert(PrixNegocie(Produit,Prix2));
                    assert(Possede(Produit))
                END
            ELSE PrixNegocie(Produit,Prix+1)
       END.

SubPrixNegocie(Produit,Prix,Max)
    <- accept_vente(Produit,Prix)|
       BEGIN
            remove(accept_vente(Produit,Prix));
            assert(PrixNegocie(Produit,Prix));
            assert(Possede(Produit))
       END.

SubPrixNegocie(Produit,Prix,Max)
    <- dernier_prix_vente(Produit,Prix2)|
       BEGIN
            remove(dernier_prix_vente(Produit,Prix2);
            IF [Prix2 <= Max]
            THEN
                BEGIN
                    tell(Alice,accept_achat(Produit,Prix2));
                    assert(PrixNegocie(Produit,Prix2));
                    assert(Possede(Produit));
                END
            ELSE
                BEGIN
                    tell(Alice,reject_achat(Produit,Prix2));
                    assert(tropCher(Produit));
                END
       END.
```

```
SubPrixNegocie(Produit,Prix,Max)
    <- nonDispo(Produit) |
       BEGIN
            remove(nonDispo(Produit)); /* le message et la croyance
            assert(nonDispo(Produit)); /* ont ici la même syntaxe
        END.

SubPrixNegocie(Produit,Prix,Max)
    <- reject_vente(Produit,Prix) |
        BEGIN
            remove(reject_vente(Produit,Prix));
            assert(tropCher(Produit));
        END.
```

8.4.2 Agent vendeur (Alice)

```
CAPABILITIES:

   { }
       assert(X)
   { X }.

   { X }
       remove(X)
   { }.

BELIEFBASE:

   nomProduit(p1,toto).
   prixMax(p1,90).
   nomProduit(p2,tutu).
   prixMax(p2,60).



GOALBASE:

   Vendu(p1).
   Vendu(p2).
```

```
RULEBASE:

   Vendu(Produit)
       <- PrixMin(Produit,Min) AND proposition_achat(Produit,Prix)  |
          BEGIN
               remove(proposition_achat(Produit,Prix);
               IF [Prix <= Min]
               THEN
                   BEGIN
                       tell(Bob,accept_vente(Produit,Prix);
                       assert(Vendu(Produit))
                   END
               ELSE PrixNegocie(Produit,Prix*4/3)
          END.

   PrixNegocie(Produit,Prix) <- prixMin(Produit,Min) |
       IF [Prix > Min]
       THEN tell(Bob,propose_vente(Produit,Prix))
       ELSE
           BEGIN
               tell(Bob,dernier_prix_vente(Produit,Prix));
               SubPrixNegocie(Produit,Prix,Min)
           END.

    SubPrixNegocie(Produit,Prix,Min)
       <- proposition_achat(Produit,Prix2)|
          BEGIN
               remove(proposition_achat(Produit,Prix2));
               IF [Prix2 >= Min] THEN
                   BEGIN
                       tell(Bob,accept_vente(Produit,Prix2));
                       assert(PrixNegocie(Produit,Prix2));
                       assert(Vendu(Produit))
                   END
               ELSE PrixNegocie(Produit,Prix-1)
          END.
```

```
SubPrixNegocie(Produit,Prix,Min)
    <- accept_achat(Produit,Prix)|
       BEGIN
            remove(accept_achat(Produit,Prix));
            assert(PrixNegocie(Produit,Prix));
            assert(Vendu(Produit))
       END.

SubPrixNegocie(Produit,Prix,Min)
    <- dernier_prix_achat(Produit,Prix2)|
       BEGIN
            remove(dernier_prix_achat(Produit,Prix2);
            IF [Prix2 >= Min]
            THEN
                BEGIN
                    tell(Bob,accept_vente(Produit,Prix2));
                    assert(PrixNegocie(Produit,Prix2));
                    assert(Vendu(Produit));
                END
            ELSE
                BEGIN
                    tell(Bob,reject_vente(Produit,Prix2));
                    assert(nonVendable(Produit));
                END
       END.

SubPrixNegocie(Produit,Prix,Min)
    <- reject_achat(Produit,Prix) |
        BEGIN
            remove(reject_achat(Produit,Prix));
            assert(nonVendable(Produit));
        END.

<- proposition_achat(Produit,Prix)
   AND [NOT nomProduit(Produit,Nom)] |
   BEGIN
        remove(proposition_achat(Produit,Prix));
        tell(Bob,non_dispo(Produit))
   END.
```

Conclusion

Comme nous l'avons dit dans l'introduction, le paradigme orienté-agent permettrait de résoudre certains des problèmes actuels de l'ingénierie logicielle, en alliant des mécanismes de communication, de coordination et de négociation, à des techniques issues de l'intelligence artificielle.

Toutefois, il est intéressant de se demander si cet avantage "théorique" des SMA est réellement mis en oeuvre dans les différentes méthodes et outils présentés tout au long de ce mémoire, et également si le développement de SMA grâce à ces méthodes et outils est suffisamment systématique pour être applicable au niveau de l'entreprise.

Du point de vue des méthodologies, nous avons vu qu'aucune ne paraît s'imposer comme étant la panacée. En effet, la méthodologie AAII présente l'avantage de se baser sur le paradigme BDI, ce qui lui permet d'être implémentable dans des langages tels que 3APL, AgentSpeak(L), ... Toutefois, la modélisation d'interactions complexes est difficilement réalisable, dû au manque de normalisation. La méthodologie Gaia quant à elle fournit un modèle d'interactions, mais celui-ci ne permet pas de modéliser des protocoles de coopération et de négociation complexes. De plus, les modèles fournis par cette méthodologie sont plus généraux, et par là-même plus difficile à traduire dans un langage de programmation orienté-agent. Finalement, la méthodologie MAS-CommonKADS fournit des modèles permettant de capturer aussi bien les interactions complexes entre agents que leur "comportement intelligent". Malheureusement, comme nous l'avons dit précédemment, aucun langage de programmation de haut-niveau ne permet d'implémenter une spécification obtenue grâce à MAS-CommonKADS. Nous pouvons donc constater qu'aucune méthodologie présentée ici n'a réussi à modéliser de manière efficace les interactions entre agents, tout en fournissant des modèles aisément implémentables. Toutefois, il est important de préciser que les langages de programmation orientés-agent actuels ne permettent pas encore la spécification d'interactions complexes, ce qui n'encourage évidemment pas les méthodologies à fournir des modèles pour celles-ci.

Du point de vue des langages de programmation orientés-agent, nous avons vu que leur utilisation dans l'implémentation d'agents est minime. 3APL, Congolog et AgentSpeak(L) ne sont encore encore qu'au stade de la recherche et du développement. En outre, le langage JAVA est sans aucun doute le langage le plus usité dans la programmation d'agents. Toutefois ce langage entraîne un problème au niveau conceptuel car il est orienté objet et non agent, une transposition est donc nécessaire. L'utilité principale de 3APL et AgentSpeak(L) est de permettre l'exécution de spécifications BDI. Ces deux langages basés sur des règles s'exécutent en temps réel et permettent d'implementer de manière intuitive le comportement "intelligent" des agents. Congolog, quant à lui, est basé sur le calcul des situations et a l'avantage de pouvoir gérer de l'information incomplète et d'affiner la modélisation du comportement de l'agent grâce aux concepts d'interruption et de prioritisation, il ne peut toutefois pas s'exécuter en temps réel. Cependant, ces langages attachent peu d'importance à la normalisation des interactions entre agents, même si différentes initiatives commence à voir le jour tel que FIPA ACL. Ces langages semblent être plus axés sur l'implémentation du comportement individuel des agents plutôt que sur le système dans son ensemble. Les agents devraient mettre en oeuvre des processus de communication, de collaboration et de négociation plus complexes afin de réaliser l'objectif global du système.

Néanmoins, malgré les différents problèmes présentés ci-dessus, nous avons montré que le développement systématique d'un système multi-agents au moyen de la méthodologie AAII et du langage de programmation 3APL est possible, bien que ce système soit relativement simple. Ceci tend à montrer l'ingénierie logicielle orientée-agent est méthodologiquement possible, il reste donc aux méthodes et outils présentés ici d'intégrer parfaitement les différents mécanismes qui font la puissance théorique du paradigme orienté-agent.

En ce qui concerne les perspectives d'avenir, il nous paraît indispensable que les langages de programmation orientés-agent et les formalismes adoptent une norme telle que FIPA ACL afin de permettre la coopération d'agents hétérogènes. De plus, l'intégration de la notion d'attitude sociale au sein du paradigme BDI nous paraît essentiel afin de disposer d'un outil d'abstraction sur le comportement global d'un système. Finalement, la popularité du paradigme orienté-agent dépendra également de l'émergence d'un langage de formulation standard qui puisse capturer tous les concepts essentiels aux systèmes multi-agents, comme l'est UML pour le paradigme orienté-objet.

En guise de conclusion,le paradigme orienté-agent permettrait de disposer d'un nouveau point de vue très intéressant pour la conception de logiciels, mais il faut néanmoins laisser le temps aux outils et aux méthodes qui le caractérisent d'arriver à maturité.

Bibliographie

[1] B. Bauer, J. P. Müller, and J. Odell, Agent UML : A formalism for specifying multiagent software systems, Agent-Oriented Software Engineering - Proceedings of the First International Workshop (AOSE-2000) (M. Wooldridge P. Ciancarini, ed.), Springer-Verlag, 2000.

[2] G. Booch, Object-oriented analysis and design with applications, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.

[3] Brahim Chaib-draa, Agents et systèmes multiagents (IFT 64881A) notes de cours département d'informatique faculté des sciences et de génie, université de Laval, 1999.

[4] B. Chandrasekaran, T. R. Johnson, and J. W. Smith, Task structure analysis for knowledge modelling, Communications of the ACM, vol. 35, 1992, pp. 124–137.

[5] W. J. Clancey, Heuristic classification, Artificial Intelligence, vol. 27, 1985, pp. 289–350.

[6] P. R. Cohen and H. J. Levesque, Intention = choice + commitment, Proc. of AAAI-87 (Seattle, WA), 1987, pp. 410–415.

[7] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, Object-oriented development : The FUSION method, Prentice Hall International : Hemel Hempstead, England, 1994.

[8] M. d'Inverno and M. Luck, Engineering AgentSpeak(L), a formal computational model, 1998.

[9] Mark d'Inverno, Koen Hindriks, and Michael Luck, A formal architecture for the 3APL agent programming language.

[10] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge, A formal specification of dMARS, Agent Theories, Architectures, and Languages, 1997, pp. 155–176.

[11] E. A. Emerson and J. Srinivasan, Branching time temporal logic, Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency (J. W. de Bakker,

W.-P. de Roever, and G. Rozenberg, eds.), Lecture Notes in Computer Science, vol. 354, Springer, June 1988, pp. 123–172.

[12] Jacques Ferber, Les systèmes multi-agents, vers une intelligence collective, InterEditions, Paris, 1995.

[13] ———, Multi-agent systems, an introduction to distributed artificial intelligence, InterEditions, Paris, 1999.

[14] M. R. Genereth and N. Nilson, Logical foundations of artificial intelligence, Morgan Kaufmann Publishers, San Mateo, CA, 1987.

[15] Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque, Congolog, a concurrent programming language based on the situation calculus, Artificial Intelligence **121** (2000), no. 1-2, 109–169.

[16] Norbert Glaser, Contribution to knowledge modelling in a multi-agent framework (the CoMoMAS approach), Ph.D. thesis, L'Université Henri Poincaré, Nancy I, France, Novembre 1996.

[17] Koen Hindriks, Frank de Boer, Wiebe van der Hoek, and John-Jules Meyer, Failure, monitoring and recovery in the agent language 3APL, AAAI 1998 Fall Symposium on Cognitive Robotics (Giuseppe De Giacomo, ed.), 1998, pp. 68–75.

[18] Koen Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer, A Formal Embedding of AgentSpeak(L) in 3APL.

[19] Koen Hindriks, Mark d'Inverno, and Michael Luck, An architecture for 3APL.

[20] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer, Agent programming in 3APL, Autonomous Agents and Multi-Agent Systems **2** (1999), no. 4, 357–401.

[21] J. Hintikka, Knowledge and belief, Cornell University Press, Ithaca, NY, 1962.

[22] http ://java.sun.com.

[23] http ://www.cs.sunysb.edu/ sbprolog/xsb-page.html.

[24] http ://www.cs.toronto.edu/cogrobo/Papers/kr02-semdelindigolog.pdf.

[25] http ://www.cs.uu.nl/docs/vakken/crob/pmain.html.

[26] C. Iglesias, M. Garijo, J. Gonz, and l Velasco, A methodological proposal for multiagent systems development extending CommonKADS, 1996.

[27] C. A. Iglesias, J. C. González, and J. R. Velasco, MIX : A general purpose multiagent architecture, Intelligent Agents II — Agent Theories, Architectures, and Languages (LNAI 1037) (M. Wooldridge, J.-P. Müller, and M. Tambe, eds.), Springer-Verlag : Heidelberg, Germany, 1996, pp. 251–266.

[28] Carlos Iglesias, Mercedes Garrijo, and José Gonzalez, A survey of agent-oriented methodologies, Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98) (Jörg Müller, Munindar P. Singh, and Anand S. Rao, eds.), vol. 1555, Springer-Verlag : Heidelberg, Germany, 1999, pp. 317–330.

[29] Carlos Argel Iglesias, Mercedes Garijo, Jose Centeno-Gonzalez, and Juan R. Velasco, Analysis and design of multiagent systems using MAS-common KADS, Agent Theories, Architectures, and Languages, 1997, pp. 313–327.

[30] ITU-T, CCITT specification and description language (SDL), Tech. Report Z100 (1993), ITU-T, 1994.

[31] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, Object-oriented software engineering. a use case driven approach, ACM Press, 1992.

[32] David Kinny, Michael Georgeff, and Anand Rao, A methodology and modelling technique for systems of BDI agents, Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (Eindhoven, The Netherlands) (Rudy van Hoe, ed.), 1996.

[33] Yves Lespérance Koen Hindriks and Hector Levesque, An embedding of ConGolog in 3APL.

[34] Y. Lesperance, H. Levesque, and R. Reiter, A situation calculus approach to modeling and programming agents, 1999.

[35] Yves Lesperance, Hector J. Levesque, Fangzhen Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl, Foundations of a logical approach to agent programming, Agent Theories, Architectures, and Languages, 1995, pp. 331–346.

[36] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl, GOLOG : A logic programming language for dynamic domains, Journal of Logic Programming **31** (1997), no. 1-3, 59–83.

[37] M. Ljungberg and A. Lucas, The OASIS air traffic management system, Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence, PRICAI '92, 1992.

[38] J. McCarthy, Ascribing mental qualities to machines, Tech. report, Stanford University AI Lab., Stanford, CA 94305, 1978.

[39] A. S. Rao and M. P. Georgeff, BDI-agents : from theory to practice, Proceedings of the First Intl. Conference on Multiagent Systems (San Francisco), 1995.

[40] Anand S. Rao, AgentSpeak(L) : BDI agents speak out in a logical computable language, Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (Eindhoven, The Netherlands) (Rudy van Hoe, ed.), 1996.

[41] Anand S. Rao and Michael P. Georgeff, Modeling rational agents within a BDI-architecture, Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91) (James Allen, Richard Fikes, and Erik Sandewall, eds.), Morgan Kaufmann publishers Inc. : San Mateo, CA, USA, 1991, pp. 473–484.

[42] E. Rudolph, J. Grabowski, and P. Graubmann, Tutorial on message sequence charts (MSC), Proceedings of FORTE/PSTV'96 Conference, 1996.

[43] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-oriented modelling and design, Prentice Hall, Englewood Cliffs, NJ, 1991.

[44] A. Th. Schreiber, B. Wielinga, R. de Hoog, H. Akkermans, and W. van de Velde, CommonKADS : A comprehensive methodology for KBS development, IEEE Expert, 1994, pp. 28–37.

[45] Y. Shoham, Agent-oriented programming, artificial intelligence volume 60, 1993.

[46] Y. Shoham and S. B. Cousins, Logics of mental attitudes in AI : A very preliminary survey, Foundations of Knowledge Representation and Reasoning (G. Lakemeyer and B. Nebel, eds.), Springer, Berlin, Heidelberg, 1994, pp. 296–309.

[47] Rudi Studer, V. Richard Benjamins, and Dieter Fensel, Knowledge engineering : Principles and methods, Data Knowledge Engineering **25** (1998), no. 1-2, 161–197.

[48] Thomas, S. R. : PLACA, an Agent Oriented Programming Language. Technical Report STAN-CS-93-1487. Standford University, 1993.

[49] G. Weiss, Multiagent systems : A modern approach to distributed artificial intelligence, 1999.

[50] Michael Wooldridge and Nicholas R. Jennings, Intelligent agents : Theory and practice, HTTP ://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.h (Hypertext version of Knowledge Engineering Review paper), 1994.

[51] Michael Wooldridge, Nicholas R. Jennings, and David Kinny, The Gaia methodology for agent-oriented analysis and design, Autonomous Agents and Multi-Agent Systems **3** (2000), no. 3, 285–312.

[52] Mike Wooldridge and P. Ciancarini, Agent-Oriented Software Engineering : The State of the Art, First Int. Workshop on Agent-Oriented Software Engineering (P. Ciancarini and M. Wooldridge, eds.), vol. 1957, Springer-Verlag, Berlin, 2000, pp. 1–28.